Programmation système en C sous Linux

Signaux, processus, threads, IPC et sockets

2e édition

Christophe Blaess

EYROLLES

Programmation système en C sous Linux

CHEZ LE MÊME ÉDITEUR

C. BLAESS. – **Scripts sous Linux**.
N°11405, 2004, 762 pages.

J.-F. BOUCHAUDY, G. GOUBET. – **Linux Administration**.
N°11505, 2004, 936 pages.

R. W. SMITH. – **Accroître les performances d'un système Linux**.
N°11430, 2004, 812 pages.

V. STANFIELD, R. W. SMITH. – **Linux**. *Guide de l'administrateur.*
N°11263, 2003, 654 pages.

C. HUNT. – **Serveurs réseau Linux**.
N°11229, 2003, 648 pages.

J.-F. BOUCHAUDY. – **TCP/IP sous Linux**.
N°11369, 2003, 866 pages.

B. BOUTHERIN, B. DELAUNAY. – **Sécuriser un réseau Linux**.
N°11445, 2004, 188 pages.

C. AULDS. – **Apache 2.0**. *Guide de l'administrateur Linux*.
N°11264, 2003, 612 pages.

B. HATCH, J. LEE, G. KURTZ. – **Halte aux hackers Linux**.
N°25487, 2003, 660 pages.

G. BRIARD. – **Oracle9*i* sous Linux**.
N°11026, 2001, 840 pages.

P. FICHEUX. – **Linux embarqué**.
N°11024, 2002, 326 pages.

B. WARD. – **SOS dépannage Linux**.
N°9253, 2001, 258 pages.

Programmation système en C sous Linux

Signaux, processus, threads, IPC et sockets

2e édition

Christophe Blaess

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

DANGER
LE PHOTOCOPILLAGE TUE LE LIVRE

Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2000, 2005, ISBN : 2-212-11601-2

Avant-propos

La dynamique des logiciels libres en général et du système Gnu/Linux en particulier a commencé à prendre de l'importance il y a quelques années, et son succès ne s'est pas démenti depuis. Le scepticisme que l'on rencontrait encore parfois à l'encontre de Linux a fortement diminué, et on trouve de plus en plus d'applications professionnelles de ce système, notamment dans les domaines industriels et scientifiques, ainsi que pour les services réseau.

Bien au-delà du « pingouin aux œufs d'or » dont trop de gens aimeraient profiter, il s'agit en réalité d'un phénomène technologique particulièrement intéressant. La conception même du noyau Linux ainsi que celle de tout l'environnement qui l'accompagne sont des éléments passionnants pour le programmeur. La possibilité de consulter les sources du système d'exploitation, de la bibliothèque C ou de la plupart des applications représente une richesse inestimable non seulement pour les passionnés qui désirent intervenir sur le noyau, mais également pour les développeurs curieux de comprendre les mécanismes intervenant dans les programmes qu'ils utilisent.

Dans cet ouvrage, j'aimerais communiquer le plaisir que j'éprouve depuis plusieurs années à travailler quotidiennement avec un système Linux. Je me suis trouvé professionnellement dans divers environnements industriels utilisant essentiellement des systèmes Unix classiques. L'emploi de PC fonctionnant sous Linux nous a permis de multiplier le nombre de postes de travail et d'enrichir nos systèmes en créant des stations dédiées à des tâches précises (filtrage et diffusion de données, postes de supervision…), tout en conservant une homogénéité dans les systèmes d'exploitation des machines utilisées.

Ce livre est consacré à Linux en tant que noyau, mais également à la bibliothèque Gnu GlibC, qui lui offre toute sa puissance applicative. On considérera que le lecteur est à l'aise avec le langage C et avec les commandes élémentaires d'utilisation du système Linux. Dans les programmes fournis en exemple, l'effort a porté sur la lisibilité du code source plutôt que sur l'élégance du codage. Les ouvrages d'initiation au langage C sont nombreux ; on conseillera l'indispensable [KERNIGHAN 1994], ainsi que l'excellent cours [CASSAGNE 1998], disponible librement sur Internet. En ce qui concerne l'installation et l'utilisation de Linux, on se tournera par exemple vers [WELSH 2003].

Le premier chapitre présentera rapidement les concepts et les outils nécessaires au développement sous Linux. Les utilitaires ne seront pas détaillés en profondeur, on se reportera aux documentations les accompagnant (pages de manuels, fichiers `info`, etc.).

Nous aborderons ensuite la programmation proprement dite avec Linux et la GlibC. Nous pouvons distinguer cinq parties successives :

- Les chapitres 2 à 11 sont plus particulièrement orientés vers l'exécution des programmes. Nous y verrons les identifications des processus, l'accès à l'environnement, le lancement et

l'arrêt d'un logiciel. Nous traiterons également des signaux, en examinant les extensions temps-réel, des entrées-sorties simples et de l'ordonnancement des processus. Nous terminerons cette partie par une présentation des threads Posix.

- La deuxième partie sera consacrée à la mémoire, tant au niveau des mécanismes d'allocation et de libération que de l'utilisation effective des blocs ou des chaînes de caractères. Cette partie recouvrira les chapitres 13 à 17 et se terminera par l'étude des traitements avancés sur les blocs de mémoire, comme les expressions régulières ou le cryptage DES.
- Nous aurons ensuite une série de chapitres consacrés aux fichiers. Les chapitres 18 et 19 serviront à caractériser les descripteurs de fichiers et les flux, puis les chapitres 20 à 22 décriront les opérations sur les répertoires, les attributs des fichiers et les bases de données disponibles avec la GlibC.
- Les chapitres 23 à 27 peuvent être considérés comme traitant des données elles-mêmes, aussi bien les types spécifiques comme les caractères étendus que les fonctions mathématiques, les informations fournies par le système d'exploitation ou l'internationalisation des programmes.
- Enfin, la dernière partie de ce livre mettra l'accent sur les communications, tout d'abord entre processus résidant sur le même système, avec les mécanismes classiques et les IPC Système V. Nous verrons ensuite une introduction à la programmation réseau et à l'utilisation des terminaux pour configurer des liaisons série. Dans cette partie qui s'étend des chapitres 28 à 33, on examinera également certains mécanismes d'entrée-sortie avancés permettant des multiplexages de canaux de communication ou des traitements asynchrones.

On remarquera que j'accorde une importance assez grande à l'appartenance d'une fonction aux normes logicielles courantes. C'est une garantie de portabilité des applications. Le standard **C Ansi** (qu'on devrait d'ailleurs plutôt nommer ISO C) est important au niveau de la syntaxe d'écriture des applications. La norme **Posix** (Posix.1, et ses extensions Posix.1b – temps réel et Posix.1c – threads) a longtemps fait figure de référence dans le domaine Unix, accompagnée d'un autre standard : **Unix 98**.

Le standard qui s'impose de nos jours est une fusion de Posix et de la norme Unix 98. Elles ont évolué ensemble pour donner naissance à **SUSv3** (*Single Unix Specifications version 3*). Non seulement ce document décrit des fonctionnalités bien respectées sur les systèmes Unix en général et Linux en particulier, mais il est en outre disponible gratuitement sur Internet, sur le site de l'association Open Group (`www.opengroup.org`).

J'aimerais profiter de la nouvelle édition de ce livre pour remercier les personnes qui m'ont soutenu lors de sa rédaction et de sa mise à jour, à commencer par mon épouse Anne-Sophie, et mes enfants Jennifer, Mina et Pierre. De nombreux lecteurs m'ont écrit pour me communiquer leurs observations et me faire part de leurs remarques, je les en remercie de tout coeur. J'ajoute un remerciement particulier à mon ami Michael Kerrisk qui m'a fourni des informations et des commentaires très judicieux.

Ris-Orangis, janvier 2005,

Christophe@Blaess.fr

http://www.blaess.fr/christophe/

Table des matières

Avant-propos V

CHAPITRE 1

Concepts et outils 1

Généralités sur le développement sous Linux 1

Outils de développement 3

Éditeurs de texte 4

Vi et Emacs 4

Éditeurs Gnome ou Kde 6

Nedit 6

Compilateur, éditeur de liens 7

Débogueur, profileur 9

Traitement du code source 14

Vérificateur de code 14

Mise en forme 15

Utilitaires divers 17

Construction d'application 18

Distribution du logiciel 19

Archive classique 20

Paquetage à la manière Red Hat 20

Environnements de développement intégré 21

Contrôle de version 21

Bibliothèques supplémentaires pour le développement 23
Interface utilisateur en mode texte 23
Développement sous X-Window 24
Les environnements Kde et Gnome 24
Conclusion 25

CHAPITRE 2
La notion de processus 27
Présentation des processus 27
Identification par le PID 29
Identification de l'utilisateur correspondant au processus 32
Identification du groupe d'utilisateurs du processus 37
Identification du groupe de processus 41
Identification de session 43
Capacités d'un processus 46
Conclusion 48

CHAPITRE 3
Accès à l'environnement 51
Variables d'environnement 51
Variables d'environnement couramment utilisées 58
Arguments en ligne de commande 60
Options simples – SUSv3 62
Options longues – Gnu 64
Sous-options 67
Exemple complet d'accès à l'environnement 68
Conclusion 74

CHAPITRE 4
Exécution des programmes 75
Lancement d'un nouveau programme 75
Causes d'échec de lancement d'un programme 81
Fonctions simplifiées pour exécuter un sous-programme 84
Conclusion 92

CHAPITRE 5
Fin d'un programme 93
Terminaison d'un programme 93
Terminaison normale d'un processus 93
Terminaison anormale d'un processus 97
Exécution automatique de routines de terminaison 100
Attendre la fin d'un processus fils 103
Signaler une erreur 114
Conclusion 122

CHAPITRE 6
Gestion classique des signaux 123
Généralités 123
Liste des signaux sous Linux 125
Signaux SIGABRT et SIGIOT 125
Signaux SIGALRM, SIGVTALRM et SIGPROF 126
Signaux SIGBUS et SIGSEGV 126
Signaux SIGCHLD et SIGCLD 127
Signaux SIGFPE et SIGSTKFLT 127
Signal SIGHUP 128
Signal SIGILL 128
Signal SIGINT 129
Signaux SIGIO et SIGPOLL 129
Signal SIGKILL 130
Signal SIGPIPE 130
Signal SIGQUIT 130
Signaux SIGSTOP, SIGCONT, et SIGTSTP 131
Signal SIGTERM 131
Signal SIGTRAP 131
Signaux SIGTTIN et SIGTTOU 131
Signal SIGURG 132
Signaux SIGUSR1 et SIGUSR2 132
Signal SIGWINCH 133
Signaux SIGXCPU et SIGXFSZ 133
Signaux temps-réel 133
Émission d'un signal sous Linux 136
Délivrance des signaux – Appels système lents 138

Réception des signaux avec l'appel système signal() 140
Conclusion 149

CHAPITRE 7
Gestion portable des signaux 151
Réception des signaux avec l'appel-système sigaction() 151
Configuration des ensembles de signaux 155
Exemples d'utilisation de sigaction() 156
Blocage des signaux 161
Attente d'un signal 165
Écriture correcte d'un gestionnaire de signaux 167
Utilisation d'un saut non local 170
Un signal particulier : l'alarme 172
Conclusion 175

CHAPITRE 8
Signaux temps-réel 177
Caractéristiques des signaux temps-réel 178
Nombre de signaux temps-réel 178
Empilement des signaux bloqués 179
Délivrance prioritaire des signaux 179
Informations supplémentaires fournies au gestionnaire 180
Émission d'un signal temps-réel 181
Traitement rapide des signaux temps-réel 189
Conclusion 191

CHAPITRE 9
Sommeil des processus et contrôle des ressources 193
Endormir un processus 193
Sommeil utilisant les temporisations de précision 199
Timers temps-réel 207
Suivre l'exécution d'un processus 212
Obtenir des statistiques sur un processus 215
Limiter les ressources consommées par un processus 217
Conclusion 225

CHAPITRE 10
Entrées-sorties simplifiées 227
Flux standard d'un processus 227
Écriture formatée dans un flux 230
Autres fonctions d'écriture formatée 238
Écritures simples de caractères ou de chaînes 241
Saisie de caractères 244
Réinjection de caractère 247
Saisie de chaînes de caractères 249
Lectures formatées depuis un flux 255
Conclusion 264

CHAPITRE 11
Ordonnancement des processus 265
États d'un processus 265
Fonctionnement multitâche, priorités 269
Modification de la priorité d'un autre processus 273
Les mécanismes d'ordonnancement sous Linux 275
Ordonnancement sous algorithme FIFO 276
Ordonnancement sous algorithme RR 277
Ordonnancement sous algorithme OTHER 277
Récapitulation 277
Temps-réel ? 278
Modification de la politique d'ordonnancement 279
Conclusion 287

CHAPITRE 12
Programmation multithread 289
Présentation 289
Implémentation 290
Création de threads 291
Attributs des threads 297
Déroulement et annulation d'un thread 301
Zones d'exclusions mutuelles 308

Attente de conditions 314
Sémaphores 319
Données privées d'un thread 324
Les threads et les signaux 325
Conclusion 329

CHAPITRE 13
Gestion de la mémoire du processus 331

Routines classiques d'allocation et de libération de mémoire 331
Utilisation de malloc() 332
Utilisation de calloc() 338
Utilisation de realloc() 340
Utilisation de free() 342
Règles de bonne conduite pour l'allocation et la libération de mémoire . . . 342
Désallocation automatique avec alloca() 344
Débogage des allocations mémoire 348
Configuration de l'algorithme utilisé par malloc() 351
Suivi intégré des allocations et des libérations 352
Surveillance automatique des zones allouées 354
Fonctions d'encadrement personnalisées 358
Conclusion 366

CHAPITRE 14
Gestion avancée de la mémoire 367

Verrouillage de pages en mémoire 367
Projection d'un fichier sur une zone mémoire 371
Protection de l'accès à la mémoire 382
Conclusion 387

CHAPITRE 15
Utilisation des blocs mémoire et des chaînes 389

Manipulation de blocs de mémoire 389
Mesures, copies et comparaisons de chaînes 395
Recherches dans une zone de mémoire ou dans une chaîne 409
Recherche dans un bloc de mémoire 409
Recherche de caractères dans une chaîne 410

Recherche de sous-chaînes 411
Analyse lexicale 414
Conclusion 417

CHAPITRE 16
Routines avancées de traitement des blocs mémoire 419
Utilisation des expressions rationnelles 419
Cryptage de données 426
Cryptage élémentaire 426
Cryptage simple et mots de passe 428
Cryptage de blocs de mémoire avec DES 430
Conclusion 435

CHAPITRE 17
Tris, recherches et structuration des données 437
Fonctions de comparaison 437
Recherche linéaire, données non triées 440
Recherches dichotomiques dans une table ordonnée 444
Manipulation, exploration et parcours d'un arbre binaire 450
Gestion d'une table de hachage 456
Récapitulatif sur les méthodes d'accès aux données 462
Conclusion 464

CHAPITRE 18
Flux de données 465
Différences entre flux et descripteurs 465
Ouverture et fermeture d'un flux 466
Ouverture normale d'un flux 467
Fermeture d'un flux 469
Présentation des buffers associés aux flux 469
Ouvertures particulières de flux 471
Lectures et écritures dans un flux 473
Positionnement dans un flux 477
Positionnement classique 478
Positionnement compatible Unix 98 480

Fichiers à trous 482
Problèmes de portabilité 484

Paramétrage des buffers associés à un flux 485

Type de buffers 485
Modification du type et de la taille du buffer 487

État d'un flux 490

Conclusion 491

CHAPITRE 19
Descripteurs de fichiers 493

Ouverture et fermeture d'un descripteur de fichier 493

Lecture ou écriture sur un descripteur de fichier 501

Primitives de lecture 502
Primitives d'écriture 505

Positionnement dans un descripteur de fichier 511

Manipulation et duplication de descripteurs 513

Duplication de descripteur 516
Accès aux attributs du descripteur 517
Attributs du fichier 518
Verrouillage d'un descripteur 520
Autre méthode de verrouillage 529

Conclusion 530

CHAPITRE 20
Accès au contenu des répertoires 531

Lecture du contenu d'un répertoire 531

Changement de répertoire de travail 536

Changement de répertoire racine 541

Création et suppression de répertoire 543

Suppression ou déplacement de fichiers 545

Fichiers temporaires 549

Recherche de noms de fichiers 553

Correspondance simple d'un nom de fichier 553
Recherche sur un répertoire total 555
Développement complet à la manière d'un shell 558

Descente récursive de répertoires 563
Conclusion 566

CHAPITRE 21
Attributs des fichiers 567
Informations associées à un fichier 567
Autorisation d'accès 571
Propriétaire et groupe d'un fichier 573
Taille du fichier 573
Horodatages d'un fichier 576
Liens physiques 577
Liens symboliques 579
Nœud générique du système de fichiers 582
Masque de création de fichier 585
Conclusion 586

CHAPITRE 22
Bases de données 587
Bases de données Unix DBM 589
Bases de données Unix NDBM 598
Bases de données Gnu GDBM 600
Bases de données DB Berkeley 604
Conclusion 611

CHAPITRE 23
Types de données et conversions 613
Types de données génériques 613
Catégories de caractères 614
Conversion entre catégories de caractères 617
Conversions de données entre différents types 618
Caractères étendus 625
Caractères étendus et séquences multioctets 630
Conclusion 632

CHAPITRE 24
Fonctions mathématiques 633
Fonctions trigonométriques et assimilées 634
Fonctions trigonométriques 635
Fonctions trigonométriques inverses 636
Fonctions connexes 636
Fonctions hyperboliques 637
Exponentielles, logarithmes, puissances et racines 639
Fonctions exponentielles 639
Fonctions logarithmiques 640
Puissances et racines 641
Nombres complexes 641
Calculs divers 643
Fonctions d'erreur 643
Fonction gamma 644
Fonctions de Bessel 644
Limites d'intervalles 646
Valeurs absolues et signes 648
Divisions entières, fractions, modulo 649
Infinis et erreurs 650
Valeur non numérique 650
Infinis 651
Représentation des réels en virgule flottante 653
Générateurs aléatoires 654
Générateur aléatoire du noyau 654
Générateur aléatoire de la bibliothèque C standard 655
Générateur aléatoire de la bibliothèque mathématique 656
Conclusion 658

CHAPITRE 25
Fonctions horaires 659
Horodatage et type time_t 660
Lecture de l'heure 661
Configuration de l'heure système 663
Conversions, affichages de dates et d'heures 664

Calcul d'intervalles 675
Fuseau horaire 676
Conclusion 678

CHAPITRE 26
Accès aux informations du système 679
Groupes et utilisateurs 679
Fichier des groupes 679
Fichier des utilisateurs 682
Fichier des interpréteurs shell 684
Nom d'hôte et de domaine 685
Nom d'hôte 685
Nom de domaine 686
Identifiant d'hôte 686
Informations sur le noyau 686
Identification du noyau 686
Informations sur l'état du noyau 688
Système de fichiers 689
Caractéristiques des systèmes de fichiers 690
Informations sur un système de fichiers 695
Montage et démontage des partitions 697
Journalisation 697
Journal utmp 698
Fonctions X/Open 701
Journal wtmp 702
Journal syslog 703
Conclusion 706

CHAPITRE 27
Internationalisation 709
Principe 710
Catégories de localisations disponibles 710
Traduction de messages 714
Catalogues de messages gérés par catgets() 714
Catalogues de messages Gnu GetText 718

Configuration de la localisation 721
Localisation et fonctions de bibliothèques 724
Localisation et fonctions personnelles 729
Informations numériques et monétaires avec localeconv() 729
Informations complètes avec nl_langinfo() 732
Conclusion 735

CHAPITRE 28
Communications classiques entre processus 737
Les tubes 738
Les tubes nommés 749
Conclusion 755

CHAPITRE 29
Communications avec les IPC Système V 757
Principes généraux des IPC Système V 757
Obtention d'une clé 758
Ouverture de l'IPC 759
Contrôle et paramétrage 759
Files de messages 760
Mémoire partagée 770
Sémaphores 772
Conclusion 782

CHAPITRE 30
Entrées-sorties avancées 783
Entrées-sorties non bloquantes 783
Attente d'événements – Multiplexage d'entrées 788
Distribution de données – Multiplexage de sorties 794
Entrées-sorties asynchrones avec fcntl() 796
Entrées-sorties asynchrones Posix.1b 797
Écritures synchronisées 807
Conclusion 810

CHAPITRE 31
Programmation réseau 813
Réseaux et couches de communication 813
Protocoles 817
Ordre des octets 820
Services et numéros de ports 822
Manipulation des adresses IP 827
Noms d'hôtes et noms de réseaux 833
Gestion des erreurs 837
Conclusion 838

CHAPITRE 32
Utilisation des sockets 839
Concept de socket 839
Création d'une socket 839
Affectation d'adresse 842
Mode connecté et mode non connecté 845
Attente de connexions 846
Demander une connexion 851
Fermeture d'une socket 855
Recevoir ou envoyer des données 857
Accès aux options des sockets 863
Programmation d'un démon ou utilisation de inetd 867
Conclusion 870

CHAPITRE 33
Gestion du terminal 873
Définition des terminaux 873
Configuration d'un terminal 875
Membre c_iflag de la structure termios 878
Membre c_oflag de la structure termios 879
Membre c_cflag de la structure termios 879
Membre c_lflag de la structure termios 880
Membre c_cc[ ] de la structure termios 881

Basculement du terminal en mode brut 883
Connexion à distance sur une socket 887
Utilisation d'un pseudo-terminal 890
Configuration d'un port série RS-232 897
Conclusion 905

ANNEXE 1
Tables Ascii et ISO 8859-1/15 907

ANNEXE 2
Fonctions et appels-système étudiés 909

ANNEXE 3
Bibliographie 935
Standards 935
Livres et articles 935
Documents informatiques 937

Index 941

1

Concepts et outils

Ce chapitre a pour but de présenter les principes généraux de la programmation sous Linux, ainsi que les outils disponibles pour réaliser des applications. Nous nous concentrerons bien entendu sur le développement en C « pur », mais nous verrons aussi des utilitaires et des bibliothèques permettant d'étendre les possibilités de la bibliothèque GlibC.

Nous ne présenterons pas le détail des commandes permettant de manipuler les outils décrits mais plutôt leurs rôles, pour bien comprendre comment s'organise le processus de développement.

Généralités sur le développement sous Linux

Dans une machine fonctionnant sous Linux, de nombreuses couches logicielles sont empilées, chacune fournissant des services aux autres. Il est important de comprendre comment fonctionne ce modèle pour savoir où une application viendra s'intégrer.

La base du système est le **noyau**, qui est le seul élément à porter véritablement le nom « Linux ». Le noyau est souvent imaginé comme une sorte de logiciel mystérieux fonctionnant en arrière-plan pour surveiller les applications des utilisateurs, mais il s'agit avant tout d'un ensemble cohérent de routines fournissant des services aux applications, en s'assurant de conserver l'intégrité du système. Pour le développeur, le noyau est surtout une interface entre son application, qui peut être exécutée par n'importe quel utilisateur, et la machine physique dont la manipulation directe doit être supervisée par un dispositif privilégié.

Le noyau fournit donc des points d'entrée, qu'on nomme « appels-système », et que le programmeur invoque comme des sous-routines offrant des services variés. Par exemple l'appel-système `write()` permet d'écrire des données dans un fichier. L'application appelante n'a pas besoin de savoir sur quel type de système de fichiers (`ext3`, `reiserfs`, `vfat`…) l'écriture se fera. L'envoi des données peut même avoir lieu de manière transparente dans un tube de communication entre applications ou vers un client distant connecté par réseau. Seul le noyau s'occupera de la basse besogne consistant à piloter les contrôleurs de disque, gérer la mémoire ou coordonner le fonctionnement des périphériques comme les cartes réseau.

Il existe une centaine d'appels-système sous Linux. Ils effectuent des tâches très variées, allant de l'allocation mémoire aux entrées-sorties directes sur un périphérique, en passant par la gestion du système de fichiers, le lancement d'applications ou la communication réseau.

L'utilisation des appels-système est en principe suffisante pour écrire n'importe quelle application sous Linux. Toutefois, ce genre de développement serait particulièrement fastidieux, et la portabilité du logiciel résultant serait loin d'être assurée. Les systèmes Unix compatibles avec la norme SUSv3 offrent normalement un jeu d'appels-système commun, assurant ainsi une garantie de compatibilité minimale. Néanmoins, cet ensemble commun est loin d'être suffisant dès qu'on dépasse le stade d'une application triviale.

Il existe donc une couche supérieure avec des fonctions qui viennent compléter les appels-système, permettant ainsi un développement plus facile et assurant également une meilleure portabilité des applications vers les environnements non SUSv3. Cette interface est constituée par la **bibliothèque C**.

Cette bibliothèque regroupe des fonctionnalités complémentaires de celles qui sont assurées par le noyau, par exemple toutes les fonctions mathématiques (le noyau n'utilise jamais les nombres réels). La bibliothèque C permet aussi d'encapsuler les appels-système dans des routines de plus haut niveau, qui sont donc plus aisément portables d'une machine à l'autre. Nous verrons à titre d'exemple dans le chapitre 18 que les descripteurs de fichiers fournis par l'interface du noyau restent limités à l'univers Unix, alors que les flux de données qui les encadrent sont portables sur tout système implémentant une bibliothèque ISO C, tout en ajoutant d'ailleurs des fonctionnalités importantes. Les routines proposées par la bibliothèque C (par exemple `malloc()` et toutes les fonctions d'allocation mémoire) sont aussi un moyen de faciliter la tâche du programmeur en offrant une interface de haut niveau pour des appels-système plus ardus, comme `sbrk()`.

Il y a eu plusieurs bibliothèques C successivement utilisées sous Linux. Les versions 1 à 4 de la **libc** Linux étaient principalement destinées aux programmes exécutables utilisant le format « `a.out` ». La version 5 de la *libc* a représenté une étape importante puisque le standard exécutable est devenu le format `elf`, beaucoup plus performant que le précédent. À partir de la version 2.0 du noyau Linux, toutes les distributions ont basculé vers une autre version de bibliothèque, la **GlibC**, issue du projet Gnu. Elle est parfois nommée – abusivement – *libc* 6. Au moment de la rédaction de ce texte, la version utilisée de la GlibC est la 2.3.2, mais elle est évolue régulièrement. Toutefois, les fonctionnalités que nous étudierons ici resteront normalement immuables pendant longtemps.

> Pour connaître votre version de bibliothèque C, exécutez depuis le shell le fichier `/lib/libc.so.6`, le numéro de noyau est visible avec la commande `uname -a`.

La bibliothèque GlibC 2 est très performante. Elle se conforme de manière précise aux standards actuels comme SUSv3, tout en offrant des extensions personnelles innovantes. Le développeur sous Linux dispose donc d'un environnement de qualité, permettant aussi bien l'écriture d'applications portables que l'utilisation d'extensions Gnu performantes. La disponibilité du code source de la GlibC 2 rend également possible la transposition d'une particularité Gnu vers un autre système en cas de portage du logiciel.

En fait la richesse de la bibliothèque GlibC rend plus facile le portage d'une application depuis un autre environnement vers Linux que dans le sens inverse. Le défaut – résultant de cette richesse – est la taille de la bibliothèque C. Elle est souvent trop volumineuse pour être employée dans des environnements restreints et des systèmes embarqués. On lui préfère alors d'autres bibliothèques C (comme DietLibc, ou uClibc), moins riches mais sensiblement plus légères.

Les fonctions de la bibliothèque GlibC et les appels-système représentent un ensemble minimal de fonctionnalités indispensables pour le développement d'applications. Ils sont pourtant très limités en termes d'interface utilisateur. Aussi plusieurs bibliothèques de fonctions ont-elles été créées pour rendre le dialogue avec l'utilisateur plus convivial. Ces bibliothèques sortent du cadre de ce livre, mais nous en citerons quelques-unes à la fin de ce chapitre.

Le programmeur retiendra donc que nous décrirons ici deux types de fonctions, les appels-système, implémentés par le noyau et offrant un accès de bas niveau aux fonctionnalités du système, et les routines de bibliothèques, qui peuvent compléter les possibilités du noyau, mais aussi l'encadrer pour le rendre plus simple et plus portable. L'invocation d'un appel-système est une opération assez coûteuse, car il est nécessaire d'assurer une commutation du processus en mode noyau avec toutes les manipulations que cela impose sur les registres du processeur. L'appel d'une fonction de bibliothèque au contraire est un mécanisme léger, équivalent à l'appel d'une sous-routine du programme (sauf bien entendu quand la fonction de bibliothèque invoque elle-même un appel-système).

Pour obtenir plus de précisions sur le fonctionnement du noyau Linux, on pourra se reporter à [LOVE 2003] *Linux Kernel Development*, ou directement aux fichiers source de Linux

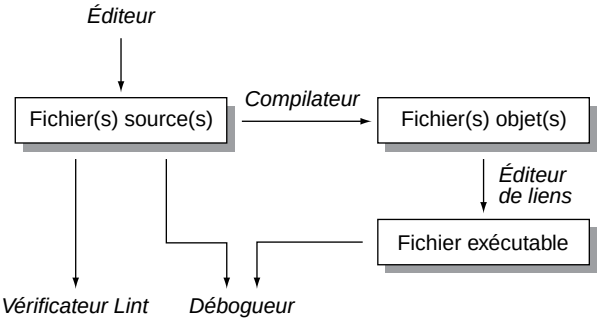
Pour des détails sur l'implémentation des systèmes Unix, l'ouvrage [BACH 1989] *Conception du système Unix* est un grand classique, ainsi que [TANENBAUM 1997] *Operating Systems, Design and Implementation*.

Outils de développement

Le développement en C sous Linux comme sous la plupart des autres systèmes d'exploitation met en œuvre principalement cinq types d'utilitaires :

- *L'éditeur de texte*, qui est à l'origine de tout le processus de développement applicatif. Il nous permet de créer et de modifier les fichiers source.
- *Le compilateur*, qui permet de passer d'un fichier source à un fichier objet. Cette transformation se fait en réalité en plusieurs étapes grâce à différents composants (préprocesseur C, compilateur, assembleur), mais nous n'avons pas besoin de les distinguer ici.
- *L'éditeur de liens*, qui assure le regroupement des fichiers objet provenant des différents modules et les associe avec les bibliothèques utilisées pour l'application. Nous obtenons ici un fichier exécutable.
- *Le débogueur*, qui peut alors permettre l'exécution pas à pas du code, l'examen des variables internes, etc. Pour cela, il a besoin du fichier exécutable et du code source.
- Notons également l'emploi éventuel d'*utilitaires annexes* travaillant à partir du code source, comme le vérificateur *Lint*, les enjoliveurs de code, les outils de documentation automatique, etc.

Figure 1.1
Processus de développement en C



Deux écoles de programmeurs coexistent sous Linux (et Unix en général) : ceux qui préfèrent disposer d'un environnement intégrant tous les outils de développement, depuis l'éditeur de texte jusqu'au débogueur, et ceux qui utilisent plutôt les différents utilitaires de manière séparée, configurant manuellement un fichier *Makefile* pour recompiler leur application sur un terminal Xterm, tandis que leur éditeur préféré s'exécute dans une autre fenêtre. Dans cet ouvrage, nous considérerons la situation d'un développeur préférant lancer lui-même ses outils en ligne de commande. Tous les environnements de développement intégré permettent en effet de lire un fichier *Makefile* externe ou de configurer les options du compilateur ou de l'éditeur de liens, comme nous le décrirons, alors que le cheminement inverse n'est pas forcément facile.

Nous allons décrire certaines options des différents utilitaires servant au développement applicatif, mais de nombreuses précisions supplémentaires pourront être trouvées dans [LOUKIDES 1997] *Programmer avec les outils Gnu*, ou dans la documentation accessible avec la commande `info`.

Éditeurs de texte

L'éditeur de texte est probablement la fenêtre de l'écran que le développeur regarde le plus. Il passe la majeure partie de son temps à saisir, relire, modifier son code, et il est essentiel de maîtriser parfaitement les commandes de base pour le déplacement, les fonctions de copier-coller et le basculement rapide entre plusieurs fichiers source.

Chaque programmeur a généralement son éditeur fétiche, dont il connaît les possibilités, et qu'il essaye au maximum d'adapter à ses préférences. Il existe deux champions de l'édition de texte sous Unix, **Vi** d'une part et **Emacs** de l'autre. Ces deux logiciels ne sont pas du tout équivalents, mais ont chacun leurs partisans et leurs détracteurs.

Vi et Emacs

Emacs est théoriquement un éditeur de texte, mais des possibilités d'extension par l'intermédiaire de scripts Lisp en ont fait une énorme machine capable d'offrir l'essentiel des commandes dont un développeur peut rêver.

Vi est beaucoup plus léger, il offre nettement moins de fonctionnalités et de possibilités d'extensions que *Emacs*. Les avantages de *Vi* sont sa disponibilité sur toutes les plates-formes Unix et la possibilité de l'utiliser même sur un système très réduit pour réparer des fichiers de

configuration. La version utilisée sous Linux est nommée *vim* (mais un alias permet de le lancer en tapant simplement `vi` sur le clavier).

Vi et *Emacs* peuvent fonctionner sur un terminal texte, mais ils sont largement plus simples à utiliser dans leur version fenêtrée X11. L'un comme l'autre nécessitent un apprentissage. Il existe de nombreux ouvrages et didacticiels pour l'utilisation de ces éditeurs performants.

Figure 1.2
Vi sous X11

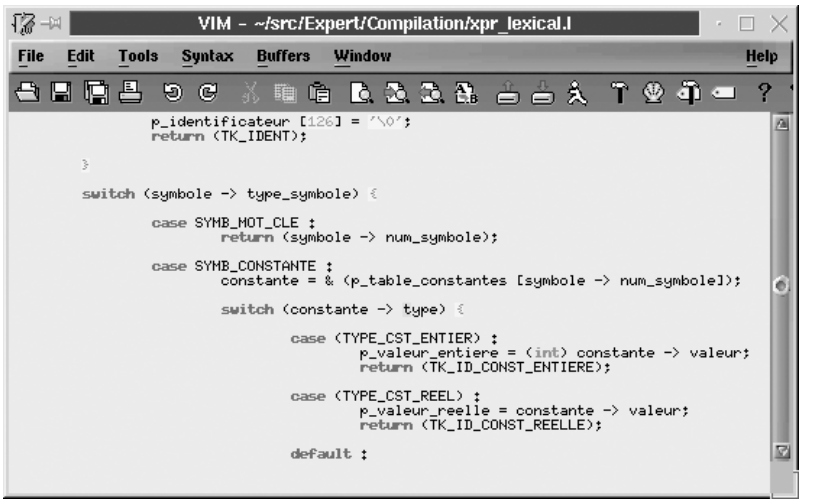
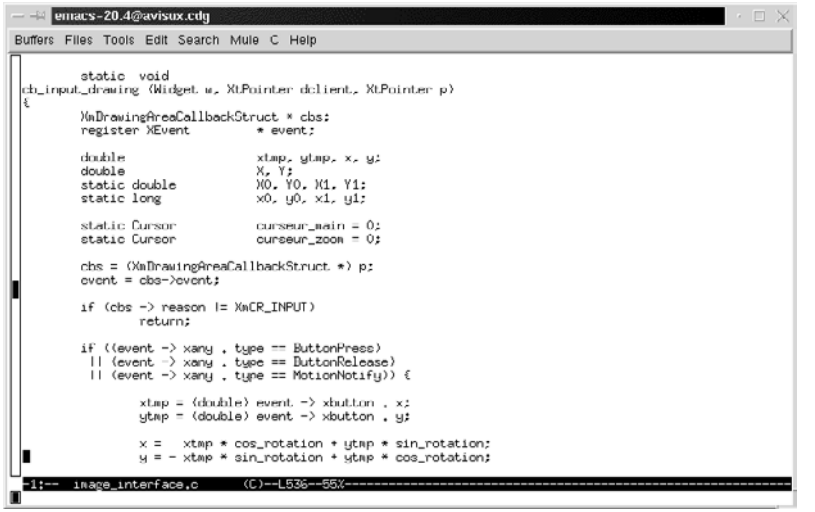


Figure 1.3
Emacs sous X11



Éditeurs Gnome ou Kde

Les deux environnements graphiques les plus en vogue actuellement, **Gnome** et **Kde**, proposent tous deux un éditeur de texte parfaitement incorporé dans l'ensemble des applications fournies. Malheureusement ces éditeurs ne sont pas vraiment très appropriés pour le programmeur.

Ils sont bien adaptés pour le dialogue avec le reste de l'environnement (ouverture automatique de documents depuis le gestionnaire de fichiers, accès aux données par un glissement des icônes, etc.).

En contrepartie, il leur manque les possibilités les plus appréciables pour un développeur, comme le basculement alternatif entre deux fichiers, la création de macros rapides pour répéter des commandes de formatage sur un bloc complet de texte, ou la possibilité de scinder la fenêtre en deux pour éditer une routine tout en jetant un coup d'œil sur la définition des structures en début de fichier.

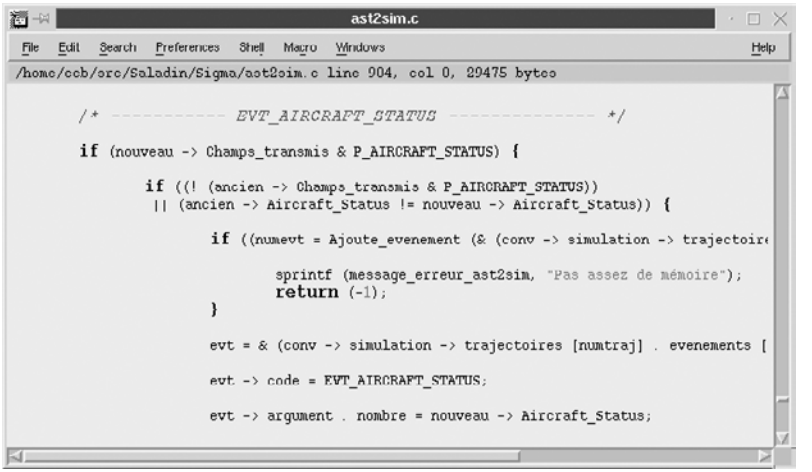
On les utilisera donc plutôt comme des outils d'appoint mais rarement pour travailler longuement sur une application.

Nedit

Comme il est impossible de citer tous les éditeurs disponibles sous Linux, je n'en mentionnerai qu'un seul, que je trouve parfaitement adapté aux besoins du développeur. L'éditeur **Nedit** est très intuitif et ne nécessite aucun apprentissage. La lecture de sa documentation permet toutefois de découvrir une puissance surprenante, tant dans la création de macros que dans le lancement de commandes externes (`make`, `spell`, `man`…), ou la manipulation de blocs de texte entiers.

Nedit est disponible – sous forme de code source ou précompilé – pour la plupart des Unix, mais n'est pas toujours installé à l'origine. L'essentiel des distributions Linux l'incluent sur leurs CD d'installation.

Figure 1.4
Nedit



Compilateur, éditeur de liens

Le compilateur C utilisé sous Linux est **gcc** (*Gnu Compiler Collection*). On peut également l'invoquer sous le nom **cc**, comme c'est l'usage sous Unix, ou **g++** si on compile du code C++. Il existe aussi une version nommée **egcs**. Il s'agit en fait d'une implémentation améliorée de gcc, effectuée par une équipe désireuse d'accélérer le cycle de développement et de mise en circulation du compilateur.

Le compilateur s'occupe de regrouper les appels aux sous-éléments utilisés durant la compilation :

- Le préprocesseur, nommé **cpp**, gère toutes les directives #define, #ifdef, #include... du code source.
- Le compilateur C proprement dit, nommé **cc1** ou **cc1plus** si on compile en utilisant la commande g++ (voire **cc1obj** si on utilise le dialecte Objective-C). Le compilateur transforme le code source prétraité en fichier contenant le code assembleur. Il est donc possible d'examiner en détail le code engendré, voire d'optimiser manuellement certains passages cruciaux (bien que ce soit rarement utile).
- L'assembleur **as** fournit des fichiers objet.
- L'éditeur de liens, nommé **ld**, assure le regroupement des fichiers objet et des bibliothèques pour fournir enfin le fichier exécutable.

Les différents outils intermédiaires invoqués par gcc se trouvent dans un répertoire situé dans l'arborescence en dessous de /usr/lib/gcc-lib/. On ne s'étonnera donc pas de ne pas les trouver dans le chemin de recherche PATH habituel du shell.

> On notera que gcc est un outil très complet, disponible sur de nombreuses plates-formes Unix. Il permet la compilation croisée, où le code pour la plate-forme cible est produit sur un environnement de développement généralement plus puissant, même si les systèmes d'exploitation et les processeurs sont différents.

Le compilateur gcc utilise des conventions sur les suffixes des fichiers pour savoir quel utilitaire invoquer lors des différentes phases de compilation. Ces conventions sont les suivantes :

Suffixe	Produit par	Rôle
.c	Programmeur	Fichier source C, sera transmis à cpp, puis à cc1.
.cc ou .C	Programmeur	Fichier source C++, sera transmis à cpp, puis à cc1plus.
.m	Programmeur	Fichier source Objective C, sera transmis à cpp, puis à cc1obj.
.h	Programmeur	Fichier d'en-tête inclus dans les sources concernées. Considéré comme du C ou du C++ en fonction du compilateur invoqué (gcc ou g++).
.i	cpp	Fichier C prétraité par cpp, sera transmis à cc1.
.ii	cpp	Fichier C++ prétraité par cpp, sera transmis à cc1plus.
.s ou .S	cc1, cc1plus, cc1obj	Fichier d'assemblage fourni par l'un des compilateurs cc1, va être transmis à l'assembleur as.
.o	as	Fichier objet obtenu après l'assemblage, prêt à être transmis à l'éditeur de liens ld pour fournir un exécutable.
.a	ar	Fichier de bibliothèque que l'éditeur de liens peut lier avec les fichiers objet pour créer l'exécutable.

En général, seules les trois premières lignes de ce tableau concernent le programmeur, car tous les autres fichiers sont transmis automatiquement à l'utilitaire adéquat. Dans le cadre de ce livre, nous ne nous intéresserons qu'aux fichiers C, même si les fonctions de bibliothèques et les appels-système étudiés peuvent très bien être employés en C++.

La plupart du temps, on invoque simplement `gcc` en lui fournissant le ou les noms des fichiers source, et éventuellement le nom du fichier exécutable de sortie, et il assure toute la transformation nécessaire. Si aucun nom de fichier exécutable n'est indiqué, `gcc` en créera un, nommé `a.out`. Ceci est simplement une tradition historique sous Unix, même si le fichier est en réalité au format actuel *elf*.

L'invocation de `gcc` se fait donc avec les arguments suivants :

- Les noms des fichiers C à compiler ou les noms des fichiers objet à lier. On peut en effet procéder en plusieurs étapes pour compiler les différents modules d'un projet, retardant l'édition des liens jusqu'au moment où tous les fichiers objet seront disponibles.
- Éventuellement des définitions de macros pour le préprocesseur, précédées de l'option `–D`. Par exemple `–D_XOPEN_SOURCE=500` est équivalent à une `directive #define _XOPEN_SOURCE 500` avant l'inclusion de tout fichier d'en-tête.
- Éventuellement le chemin de recherche des fichiers d'en-tête (en plus de `/usr/include`), précédé de l'option `–I`. Ceci est surtout utile lors du développement sous X-Window, en ajoutant par exemple `–I/usr/X11R6/include`.
- Éventuellement le chemin de recherche des bibliothèques supplémentaires (en plus de `/lib` et `/usr/lib`), précédé de l'option `–L`. Comme pour l'option précédente on utilise surtout ceci pour le développement sous X11, avec par exemple `–L/usr/X11R6/lib/`.
- Le nom d'une bibliothèque supplémentaire à utiliser lors de l'édition des liens, précédé du préfixe `-l`. Il s'agit bien du nom de la bibliothèque, et pas du fichier. Par exemple la commande `–lm` permet d'inclure le fichier `libm.so` indispensable pour les fonctions mathématiques. De même, `–lcrypt` permet d'utiliser la bibliothèque `libcrypt.so` contenant les fonctions de chiffrage DES.
- On peut préciser le nom du fichier exécutable, précédé de l'option `–o`.
- Enfin, plusieurs options simples peuvent être utilisées, dont les plus courantes sont :

Option	Argument	But
`-E`		Arrêter la compilation après le passage du préprocesseur, avant le compilateur.
`-S`		Arrêter la compilation après le passage du compilateur, avant l'assembleur.
`-c`		Arrêter la compilation après l'assemblage, laissant les fichiers objet disponibles.
`-W`	Avertissement	Valider les avertissements (*warnings*) décrits en arguments. Il en existe une multitude, mais l'option la plus couramment utilisée est `–Wall`, pour activer tous les avertissements.
`-pedantic`		Le compilateur fournit des avertissements encore plus rigoureux qu'avec `–Wall`, principalement orientés sur la portabilité du code.
`-g`		Inclure dans le code exécutable les informations nécessaires pour utiliser le débogueur. Cette option est généralement conservée jusqu'au basculement du produit en code de distribution.
`-O`	0 à 3	Optimiser le code engendré. Le niveau d'optimisation est indiqué en argument (0 = aucune). Il est déconseillé d'utiliser simultanément l'optimisation et le débogage.

Les combinaisons d'options les plus couramment utilisées sont donc :

```
gcc -Wall -pedantic -g fichier1.c -c
gcc -Wall -pedantic -g fichier2.c -c
```

qui permettent d'obtenir deux fichiers exécutables qu'on regroupe ensuite ainsi :

```
gcc fichier1.o fichier2.o -o resultat
```

On peut aussi effectuer directement la compilation et l'édition des liens :

```
gcc -Wall -pedantic -g fichier1.c fichier2.c -o resultat
```

Lorsque le code a atteint la maturité nécessaire pour basculer en version de distribution, on peut utiliser :

```
gcc -Wall -DNDEBUG -O2 fichier1.c fichier2.c -o resultat
```

La constante `NDEBUG` sert, nous le verrons dans un chapitre ultérieur, à éliminer tous le code de déboguage incorporé explicitement dans le fichier source.

Les options permettant d'ajuster le comportement de `gcc` sont tellement nombreuses que l'on pourrait y consacrer un ouvrage complet. D'autant plus que `gcc` permet le développement croisé, c'est à dire la compilation sur une machine d'une application destinée à une autre plate-forme. Ceci est particulièrement précieux pour la mise au point de programmes destinés à des systèmes embarqués par exemple, ne disposant pas nécessairement des ressources nécessaires au fonctionnement des outils de développement.

La plupart du temps nous ne nous soucierons pas de la ligne de commande utilisée pour compiler les applications, car elle se trouve incorporée directement dans le fichier de configuration du constructeur d'application `make` que nous verrons plus bas.

Débogueur, profileur

Lorsqu'une application a été compilée avec l'option –`g`, il est possible de l'exécuter sous le contrôle d'un débogueur. L'outil utilisé sous Linux est nommé **`gdb`** (*Gnu Debugger*). Cet utilitaire fonctionne en ligne de commande, avec une interface assez rébarbative. Aussi un frontal pour X-Window a été développé, nommé **`xxgdb`**. Utilisant la bibliothèque graphique *Athena Widget* du MIT, ce n'est pas non plus un modèle d'esthétique ni de convivialité.

Un autre frontal est également disponible sous Linux, nommé **ddd** (*Data Display Debugger*), plus agréable visuellement.

Le débogage d'une application pas à pas est un processus important lors de la mise au point d'un logiciel, mais ce n'est pas la seule utilisation de `gdb` et de ses frontaux. Lorsqu'un processus exécute certaines opérations interdites (écriture dans une zone non autorisée, tentative d'utilisation d'instruction illégale…) le noyau lui envoie un signal pour le tuer. Sous certaines conditions, l'arrêt de processus s'accompagne de la création d'un fichier `core` [1] sur le disque, représentant l'image de l'espace mémoire du processus au moment de l'arrêt, y compris le code exécutable.

1. Le terme `core` fait référence au noyau de fer doux se trouvant dans les tores de ferrite utilisés comme mémoire centrale sur les premières machines de l'informatique moderne. La technologie a largement évolué, mais le vocabulaire traditionnel a été conservé.

Figure 1.5
Utilisation de xxgdb

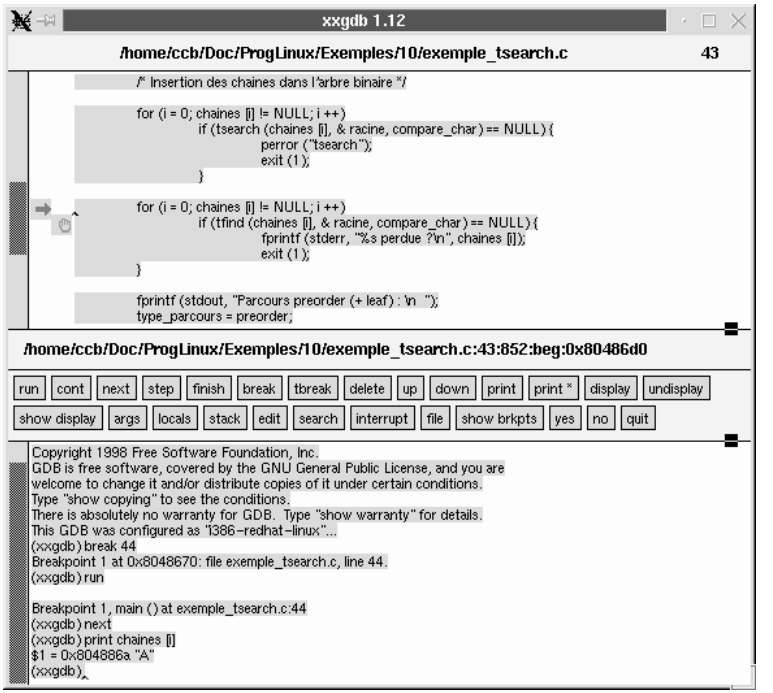
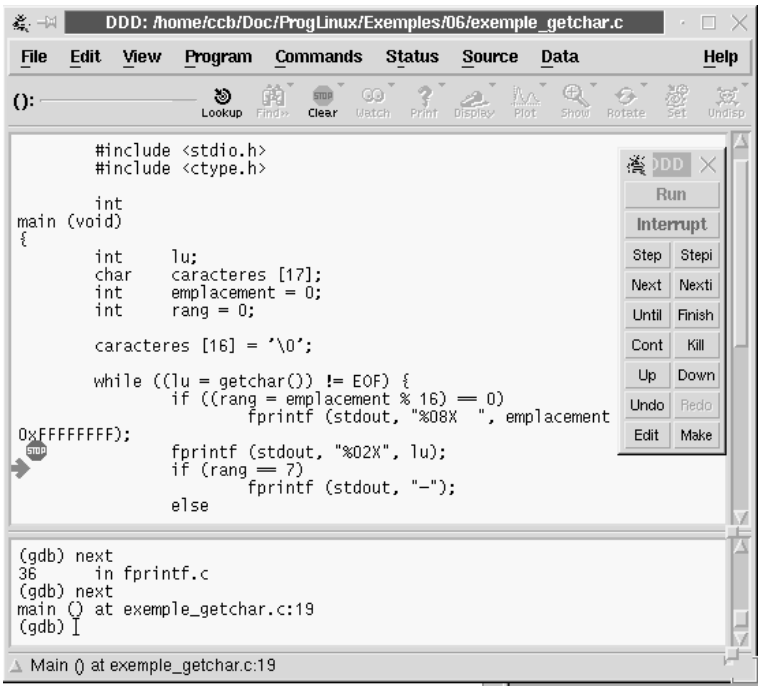


Figure 1.6
Utilisation de ddd



Le débogueur gdb est capable d'examiner ce fichier, afin de procéder à l'autopsie du processus tué. Cette analyse post-mortem est particulièrement précieuse lors de la mise au point d'un logiciel pour détecter où se produit un dysfonctionnement apparemment intempestif. De plus gdb est également capable de déboguer un processus déjà en cours de fonctionnement !

> Notons que gdb peut servir au débogage « croisé » : le débogueur fonctionne sur la station de développement tandis qu'un petit démon nommé gdbserver tourne sur la plate-forme cible (en général un système restreint) reliée par réseau ou câble série RS-232 à la station de développement. On peut ainsi déboguer à distance un programme dans son environnement d'exécution définitif, même s'il s'agit d'un système embarqué.

Dans l'informatique « de terrain », il arrive parfois de devoir analyser d'urgence les circonstances d'arrêt d'un programme au moyen de son fichier core. Ce genre d'intervention peut avoir lieu à distance, par une connexion réseau, ou par une liaison modem vers la machine où l'application était censée fonctionner de manière sûre. Dans ces situations frénétiques, il est inutile d'essayer de lancer les interfaces graphiques encadrant le débogueur, et il est nécessaire de savoir utiliser gdb en ligne de commande[1]. De toutes manières, les frontaux comme xxgdb ou ddd ne dissimulent pas le véritable fonctionnement de gdb, et il est important de se familiariser avec cet outil.

On invoque généralement le débogueur gdb en lui fournissant en premier argument le nom du fichier exécutable. Au besoin, on peut fournir ensuite le nom d'un fichier core obtenu avec le même programme.

Lors de son invocation, gdb affiche un message de présentation, puis passe en attente de commande avec un message d'invite (gdb). Pour se documenter en détail sur son fonctionnement, on tapera « help ». Le débogueur proposera alors une série de thèmes que l'on peut approfondir. Les commandes les plus courantes sont :

Commande	Rôle
list	Afficher le listing du code source.
run [argument]	Lancer le programme, qui s'exécutera jusqu'au prochain point d'arrêt.
break <ligne>	Insérer un point d'arrêt sur la ligne dont le numéro est fourni.
step	Avancer d'un pas, en entrant au besoin dans le détail des sous-routines.
next	Avancer jusqu'à la prochaine instruction, en exécutant les sous-routines sans s'arrêter.
cont	Continuer l'exécution du programme jusqu'au prochain point d'arrêt.
print <variable>	Afficher le contenu de la variable indiquée.
backtrace	Afficher le contenu de la pile, avec les invocations imbriquées des routines.
quit	Quitter le débogueur.

Il existe de très nombreuses autres commandes, comme attach <PID> qui permet de déboguer un programme déjà en cours d'exécution. Pour tout cela, on se reportera par exemple à la documentation en ligne info sur gdb.

1. Croyez-moi, lorsqu'un service d'exploitation opérationnelle vous téléphone à 22h30 car ils n'arrivent pas à relancer l'application après un redémarrage du système, on ne perd pas de temps à lancer les applications X11 au travers d'une liaison ppp par modem, et on utilise uniquement des outils en ligne de commande !

Voici un exemple de session de déboguage sur un exemple très simple du chapitre 3.

```
$ gdb exemple_argv
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
   [...]
This GDB was configured as "i386-redhat-linux"...
```

Nous commençons par demander un aperçu du listing du programme :

```
(gdb) list
1
2               #include <stdio.h>
3
4
5               int
6       main(int argc, char * argv[])
7       {
8               int i;
9
10              fprintf(stdout, "%s a reçu en argument :\n", argv[0]);
(gdb)
11              for (i = 1; i < argc; i ++)
12                      fprintf(stdout, "  %s\n", argv[i]);
13              return 0;
14      }
(gdb)
```

Nous plaçons un premier point d'arrêt sur la ligne 12 :

```
(gdb) break 12
Breakpoint 1 at 0x8048420: file exemple_argv.c, line 12.
```

Nous indiquons les arguments en ligne de commande, puis nous démarrons le programme :

```
(gdb) set args un deux trois
(gdb) run
Starting program: /home/ccb/ProgLinux/03/exemple_argv un deux trois
/home/ccb/Doc/ProgLinux/Exemples/03/exemple_argv a reçu en argument :
Breakpoint 1, main (argc=4, argv=0xbffffce4) at exemple_argv.c:12
12                      fprintf(stdout, "  %s\n", argv[i]);
```

Le programme s'étant arrêté, nous pouvons examiner ses variables :

```
(gdb) print i
$1 = 1
(gdb) print argv[i]
$2 = 0xbffffe19 "un"
```

Nous supprimons le point d'arrêt actuel, et en plaçons un nouveau sur la ligne suivante, avant de demander au programme de continuer son exécution :

```
(gdb) delete 1
(gdb) break 13
Breakpoint 2 at 0x8048450: file exemple_argv.c, line 13.
(gdb) cont
```

```
Continuing.
  un
  deux
  trois

Breakpoint 2, main(argc=4, argv=0xbffffce4) at exemple_argv.c:13
13              return 0;
```

Le programme est arrivé sur le nouveau point d'arrêt, nous pouvons le continuer en pas à pas :

```
(gdb) next
14      }
(gdb)
Program exited normally.
```

Nous quittons à présent `gdb` :

```
(gdb) quit
$
```

Il existe un autre outil important dans la phase de mise au point : le *profileur*. Cet utilitaire observe le déroulement de l'application, et enregistre dans un fichier les temps de présence dans chaque routine du programme. Il est alors facile d'analyser les goulets d'étranglement dans lesquels le logiciel passe le plus clair de son temps. Ceci permet de cibler efficacement les portions de l'application qui auront besoin d'être optimisées. Bien entendu ceci ne concerne pas tous les logiciels, loin de là, puisque la plupart des applications passent l'essentiel de leur temps à attendre les ordres de l'utilisateur. Toutefois il convient que chaque opération effectuée par le programme se déroule dans des délais raisonnables, et une simple modification d'algorithme, ou de structure de données, peut parfois permettre de réduire considérablement le temps d'attente de l'utilisateur. Ceci a pour effet de rendre l'ensemble de l'application plus dynamique à ses yeux et améliore la perception qualitative de l'ensemble du logiciel.

L'outil de profilage Gnu s'appelle **`gprof`**. Il fonctionne en analysant le fichier `gmon.out` qui est créé automatiquement lors du déroulement du processus, s'il a été compilé avec l'option `–pg` de `gcc`. Les informations fournies par `gprof` sont variées, mais permettent de découvrir les points où le programme passe l'essentiel de son temps.

On compile donc le programme à profiler ainsi :

```
$ cc -Wall -pg programme.c -o programme
```

On l'exécute alors normalement :

```
$ ./programme
$
```

Un fichier `gmon.out` est alors créé, que l'on examine à l'aide de la commande `gprof` :

```
$ gprof programme gmon.out | less
```

L'utilitaire `gprof` étant assez bavard, il est conseillé de rediriger sa sortie standard vers un programme de pagination comme `more` ou `less`. Les résultats et les statistiques obtenus sont expliqués en clair dans le texte affiché par `gprof`.

Un autre outil de suivi du programme s'appelle **`strace`**. Il s'agit d'un logiciel permettant de détecter tous les appels-système invoqués par un processus. Il observe l'interface entre le

processus et le noyau, et mémorise tous les appels, avec leurs arguments. On l'utilise simplement en l'invoquant avec le nom du programme à lancer en argument.

```
$ strace ./programme
```

Les résultats sont présentés sur la sortie d'erreur, (que l'on peut rediriger dans un fichier). Une multitude d'appels-système insoupçonnés apparaissent alors, principalement en ce qui concerne les allocations mémoire du processus.

Dans la série des utilitaires permettant d'analyser le code exécutable ou les fichiers objets, il faut également mentionner **nm** qui permet de lister le contenu d'un fichier objet, avec ses différents symboles privés ou externes, les routines, les variables, etc. Pour cela il faut bien entendu que la table des symboles du fichier objet soit disponible. Cette table n'étant plus utile lorsqu'un exécutable est sorti de la phase de déboguage, on peut la supprimer en utilisant **strip**. Cet utilitaire permet de diminuer la taille du fichier exécutable (attention à ne pas l'employer sur une bibliothèque partagée !).

Enfin, citons **objdump** qui permet de récupérer beaucoup d'informations en provenance d'un fichier objet, comme son désassemblage, le contenu des variables initialisées, etc.

Traitement du code source

Il existe toute une classe d'outils d'aide au développement qui permettent des interventions sur le fichier source. Ces utilitaires sont aussi variés que l'analyseur de code, les outils de mise en forme ou de statistiques, sans oublier les applications de manipulation de fichiers de texte, qui peuvent parfaitement s'appliquer à des fichiers sources.

Vérificateur de code

L'outil *Lint* est un grand classique de la programmation sous Unix, et son implémentation sous Linux se nomme **lclint**. Le but de cet utilitaire est d'analyser un code source C qui se compile correctement, pour rechercher d'éventuelles erreurs sémantiques dans le programme. L'appel de `lclint` peut donc être vu comme une sorte d'extension aux options `–Wall` et `–pedantic` de `gcc`.

L'invocation se fait tout simplement en appelant `lclint` suivi du nom du fichier source. On peut bien sûr ajouter des options, permettant de configurer la tolérance de `lclint` vis-à-vis des constructions sujettes à caution. Il y a environ 500 options différentes, décrites dans la page d'aide accessible avec « `lclint -help flags all` ».

L'invocation de `lclint` avec ses options par défaut peut parfois être déprimante. Je ne crois pas qu'il y ait un seul exemple de ce livre qui soit accepté tel quel par `lclint` sans déclencher au moins une page d'avertissements. Dans la plupart des cas le problème provient d'ailleurs des bibliothèques système, et il est nécessaire de relâcher la contrainte avec des options ajoutées en ligne de commande. On peut aussi insérer des commentaires spéciaux dans le corps du programme (du type `/*@null@*/`) qui indiqueront à `lclint` que la construction en question est volontaire, et qu'elle ne doit pas déclencher d'avertissement.

Cet outil est donc très utile pour rechercher tous les points litigieux d'une application. J'ai plutôt tendance à l'employer en fin de développement, pour vérifier un code source avant le passage en phase de test, plutôt que de l'utiliser quotidiennement durant la programmation. Je considère la session de vérification à l'aide de `lclint` comme une étape à part entière, à

laquelle il faut consacrer du temps, du courage et de la patience, afin d'éliminer dès que possible les bogues éventuels.

Mise en forme

Il existe un outil Unix nommé **indent**, dont une version Gnu est disponible sous Linux. Cet utilitaire est un enjoliveur de code. Ceci signifie qu'il est capable de prendre un fichier source C, et de le remettre en forme automatiquement en fonction de certaines conventions précisées par des options.

On l'utilise souvent pour des projets développés en commun par plusieurs équipes de programmeurs. Avant de valider les modifications apportées à un fichier, et l'insérer dans l'arborescence des sources maîtresses – par exemple avec `cvs` décrit plus bas – on invoque `indent` pour le formater suivant les conventions adoptées par l'ensemble des développeurs. De même lorsqu'un programmeur extrait un fichier pour le modifier, il peut appeler `indent` avec les options qui correspondent à ses préférences.

La documentation de `indent`, décrit une soixantaine d'options différentes, mais trois d'entre-elles sont principalement utiles, `-gnu` qui convertit le fichier aux normes de codage Gnu, `-kr` qui correspond à la présentation utilisée par Kernighan et Ritchie dans leur ouvrage [KERNIGHAN 1994]. Il existe aussi `–orig` pour avoir le comportement de l'utilitaire `indent` original, c'est à dire le style Berkeley. Le programme suivant va être converti dans ces trois formats :

hello.c :

```
    #include <stdio.h>

int main (int argc, char * argv [])
{
    int i;
    fprintf (stdout, "Hello world ! ");
    if (argc > 1)
    {
        fprintf (stdout, ": ");
         /* Parcours et affichage des arguments */
        for (i = 1; i < argc; i ++) fprintf (stdout, "%s ", argv [i]);
    }
    fprintf (stdout, "\n");
    return (0);
}
```

Nous demandons une mise en forme dans le style Gnu :

```
$ indent -gnu hello.c -o hello.2.c
$ cat hello.2.c
#include <stdio.h>

int
main (int argc, char *argv[])
{
  int i;
  fprintf (stdout, "Hello world ! ");
  if (argc > 1)
    {
```

```
      fprintf (stdout, ": ");
      /* Parcours et affichage des arguments */
      for (i = 1; i < argc; i++)
        fprintf (stdout, "%s ", argv[i]);
    }
  fprintf (stdout, "\n");
  return (0);
}
$
```

Voyons la conversion en style Kernighan et Ritchie :

```
$ indent -kr hello.c -o hello.3.c
$ cat hello.3.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    fprintf(stdout, "Hello world ! ");
    if (argc > 1) {
        fprintf(stdout, ": ");
        /* Parcours et affichage des arguments */
        for (i = 1; i < argc; i++)
            fprintf(stdout, "%s ", argv[i]);
    }
    fprintf(stdout, "\n");
    return (0);
}
$
```

Et finalement le style Berkeley original :

```
$ indent -orig hello.c -o hello.4.c
$ cat hello.4.c
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int             i;
    fprintf(stdout, "Hello world ! ");
    if (argc > 1) {
        fprintf(stdout, ": ");
        /*
         * Parcours et affichage des arguments
         */
        for (i = 1; i < argc; i++)
            fprintf(stdout, "%s ", argv[i]);
    }
    fprintf(stdout, "\n");
    return (0);
}
$
```

Chaque programmeur peut ainsi utiliser ses propres habitudes de mise en forme, indépendamment des autres membres de son équipe.

Utilitaires divers

L'outil **grep** est essentiel pour un programmeur, car il permet de rechercher une chaîne de caractères dans un ensemble de fichiers. Il est fréquent d'avoir à retrouver le fichier où une routine est définie, ou l'emplacement de la déclaration d'une structure par exemple. De même on a souvent besoin de rechercher à quel endroit un programme affiche un message d'erreur avant de s'arrêter. Pour toutes ces utilisations `grep` est parfaitement adapté. Sa page de manuel documente ces nombreuses fonctionnalités, et l'emploi des expressions régulières pour préciser le motif à rechercher.

Lorsque l'on désire retrouver une chaîne de caractères dans toute une arborescence, il faut le coupler à l'utilitaire **find**, en employant la commande **xargs** pour les relier. Voici à titre d'exemple la recherche d'une constante symbolique (`ICMPV6_ECHO_REQUEST` en l'occurrence) dans tous les fichiers source du noyau Linux :

```
$ cd /usr/src/linux
$ find . -type f | xargs grep ICMPV6_ECHO_REQUEST
./net/ipv6/icmp.c:  else if (type >= ICMPV6_ECHO_REQUEST &&
./net/ipv6/icmp.c: (&icmpv6_statistics.Icmp6InEchos)[type-ICMPV6_ECHO_REQUEST]++;
./net/ipv6/icmp.c:      case ICMPV6_ECHO_REQUEST:
./include/linux/icmpv6.h:#define ICMPV6_ECHO_REQUEST        128
$
```

La commande `find` recherche tous les fichiers réguliers (`-type f`) de manière récursive à partir du répertoire en cours (`.`), et envoie les résultats à `xargs`. Cet utilitaire les regroupe en une liste d'arguments qu'il transmet à `grep` pour y rechercher la chaîne demandée.

L'importance de `grep` pour un développeur est telle que les éditeurs de texte contiennent souvent un appel direct à cet utilitaire depuis une option de menu.

Lorsqu'on développe un projet sur plusieurs machines simultanément, on est souvent amené à vérifier si un fichier a été modifié et, si c'est le cas, dans quelle mesure. Ceci peut être obtenu à l'aide de l'utilitaire **diff**. Il compare intelligemment deux fichiers et indique les portions modifiées entre les deux. Cet instrument est très utile lorsqu'on reprend un projet après quelque temps et qu'on ne se rappelle plus quelle version est la bonne.

Par exemple, nous pouvons comparer les programmes hello.3.c (version Kernighan et Ritchie) et hello.4.c (version Berkeley) pour trouver leurs différences :

```
$ diff hello.3.c hello.4.c
3c3,4
< int main(int argc, char *argv[])
---
> int
> main(int argc, char *argv[])
5c6
<     int i;
---
>     int             i;
9c10,12
```

```
<       /* Parcours et affichage des arguments */
---
>       /*
>        * Parcours et affichage des arguments
>        */
$
```

Ici, `diff` nous indique une différence à la ligne 3 du premier fichier, qui se transforme en lignes 3 et 4 du second, puis une seconde variation à la ligne 5 de l'un et 6 de l'autre, ainsi qu'une dernière différence à la ligne 9, qui se transforme en 10, 11 et 12 de l'autre. On le voit, la comparaison est intelligente, `diff` essayant de se resynchroniser le plus vite possible lorsqu'il rencontre une différence. Toutefois, lorsque l'envergure d'une application augmente et que le nombre de développeurs s'accroît, il est préférable d'employer un système de contrôle de version comme `cvs`.

L'outil `diff` est aussi très utilisé dans le monde du logiciel libre et de Linux en particulier, pour créer des fichiers de différences qu'on transmet ensuite à l'utilitaire **`patch`**. Ces fichiers sont beaucoup moins volumineux que les fichiers source complets.

Ainsi, alors que les sources complètes du noyau Linux représentent une archive compressée d'environ 36 Mo (noyaux 2.6.x), les fichiers de différences publiés par Linus Torvalds tous les trois ou quatre jours mesurent en général entre 200 et 500 Ko. Pour créer un tel fichier, on utilise l'option `–u` de `diff`, suivie du nom du fichier original, puis de celui du fichier modifié. On redirige la sortie standard vers un fichier. Ce fichier peut être transmis à d'autres développeurs qui l'appliqueront sur l'entrée standard de `patch`. Ce dernier réalisera alors les modifications sur le fichier original dont une copie est présente. Tout ceci est décrit dans les pages de manuel de ces utilitaires.

Construction d'application

Dès qu'une application s'appuie sur plusieurs modules indépendants – plusieurs fichiers source C –, il est indispensable d'envisager d'utiliser les mécanismes de compilation séparée. Ainsi chaque fichier C est compilé en fichier objet `.o` indépendamment des autres modules (grâce à l'option `–c` de `gcc`), et finalement on regroupe tous les fichiers objet ensemble lors de l'édition des liens (assurée également par `gcc`).

L'avantage de ce système réside dans le fait qu'une modification apportée à un fichier source ne réclame plus qu'une seule compilation et une édition des liens au lieu de nécessiter la compilation de tous les modules du projet. Ceci est déjà très appréciable en langage C, mais devient réellement indispensable en C++, où les phases de compilation sont très longues, notamment à cause du volume des fichiers d'en-tête.

Pour ne pas être obligé de recompiler un programme source non modifié, on fait appel à l'utilitaire **`make`**. Celui-ci compare les dates de modification des fichiers source et cibles pour évaluer les tâches à réaliser. Il est aidé en cela par un fichier de configuration nommé `Makefile` (ou `makefile`, voire `GNUmakefile`), qu'on conserve dans le même répertoire que les fichiers source. Ce fichier est constitué par une série de règles du type :

```
cible : dépendances
    commandes
```

La cible indique le but désiré, par exemple le nom du fichier exécutable. Les dépendances mentionnent tous les fichiers dont la règle a besoin pour s'exécuter, et les commandes précisent comment obtenir la cible à partir des dépendances. Par exemple, on peut avoir :

```
mon_programme : interface.o calcul.o centre.o
    cc -o mon_programme interface.o c calcul.o centre.o
```

Lorsque `make` est appelé, il vérifie l'heure de modification de la cible et celles des dépendances, et peut ainsi décider de refaire l'édition des liens. Si un fichier de dépendance est absent, `make` recherchera une règle pour le créer, par exemple

```
interface.o : interface.c interface.h commun.h
    cc -Wall -pedantic -c interface.c
```

Ce système est à première vue assez simple, mais la syntaxe même des fichiers `Makefile` est assez pénible, car il suffit d'insérer un espace en début de ligne de commande, à la place d'une tabulation, pour que `make` refuse le fichier. Par ailleurs, il existe un certain nombre de règles implicites que `make` connaît, par exemple comment obtenir un fichier `.o` à partir d'un `.c`. Pour obtenir des détails sur les fichiers `Makefile`, on consultera donc la documentation Gnu.

Comme la création d'un `Makefile` peut être laborieuse, on emploie parfois des utilitaires supplémentaires, **`imake`** ou **`xmkmf`**, qui utilisent un fichier `Imakefile` pour créer le ou les fichiers `Makefile` de l'arborescence des sources. La syntaxe des fichiers `Imakefile` est décrite dans la page de manuel de `imake`.

Une autre possibilité pour créer automatiquement les fichiers `Makefile` adaptés lors d'un portage de logiciel est d'utiliser les outils Gnu **`automake`** et **`autoconf`** (voir à ce sujet la documentation `info automake`).

Distribution du logiciel

La distribution d'un logiciel sous Linux peut se faire de plusieurs manières. Tout dépend d'abord du contenu à diffuser. S'il s'agit d'un logiciel libre, le plus important est de fournir les sources du programme ainsi que la documentation dans un format le plus portable possible sur d'autres Unix. Le point le plus important ici sera de laisser l'entière liberté au destinataire pour choisir l'endroit où il placera les fichiers sur son système, l'emplacement des données de configuration, etc. On pourra consulter le document Linux *Software-Release-Practice-HOWTO*, qui contient de nombreux conseils pour la distribution de logiciels libres.

S'il s'agit de la distribution d'une application commerciale fournie uniquement sous forme binaire, le souci majeur sera plutôt de simplifier l'installation du produit, quitte à imposer certaines restrictions concernant les emplacements de l'application et des fichiers de configuration.

Pour simplifier l'installation du logiciel, il est possible de créer un script qui se charge de toute la mise en place des fichiers. Toutefois ce script devra être lancé depuis un support de distribution (CD ou disquette), ce qui nécessite une intervention manuelle de l'administrateur pour autoriser l'exécution des programmes sur un support extractible ou une copie du script dans le répertoire de l'utilisateur avant le lancement. Il est donc souvent plus simple de fournir une simple archive `tar` ou un paquetage `rpm`, et de laisser l'utilisateur les décompacter lui-même.

Archive classique

L'utilitaire **tar** (*Tape Archiver*) est employé dans le monde Unix depuis longtemps pour regrouper plusieurs fichiers en un seul paquet. À l'origine, cet outil servait surtout à copier le contenu d'un répertoire sur une bande de sauvegarde. De nos jours, on l'utilise pour créer une archive – un gros fichier – regroupant tout le contenu d'une arborescence de fichiers source.

Les conventions veulent que la distribution de l'arborescence des sources d'un projet se fasse en incluant son répertoire de départ. Par exemple si une application est développée dans le répertoire `~/src/mon_appli/` et ses sous-répertoires, il faudra que l'archive soit organisée pour qu'en la décompactant l'utilisateur se trouve avec un répertoire `mon_appli/` et ses descendants. Pour créer une telle archive, on procède ainsi :

```
$ cd ~/src
$ tar -cf mon_appli.tar mon_appli/
```

Le fichier `mon_appli.tar` contient alors toute l'archive. Pour le décompresser, on peut effectuer :

```
$ cp mon_appli.tar ~/tmp
$ cd ~/tmp
$ tar -xf mon_appli.tar
$ ls
mon_appli.tar
mon_appli/
$
```

La commande `c` de `tar` sert à créer une archive, alors que `x` sert à extraire son contenu. Le `f` précise que l'archive est un fichier dont le nom est indiqué à la suite (et pas l'entrée ou la sortie standard). On peut aussi ajouter la commande `z` entre `c` ou `x`, et `f`, pour indiquer que l'archive doit être compressée ou décompressée en invoquant **gzip**., ou « `j` » pour la compresser ou décompresser avec **bzip2**.

Lorsqu'on désire fournir un fichier d'installation regroupant un exécutable, à placer par exemple dans `/usr/local/bin`, et des données se trouvant dans `/usr/local/lib/...`, ainsi qu'un fichier d'initialisation globale dans `/etc`, l'emploi de `tar` est toujours possible mais moins commode. Dans ce cas, il faut créer l'archive à partir de la racine du système de fichiers en indiquant uniquement les fichiers à incorporer. L'extraction sur le système de l'utilisateur devra aussi être réalisée à partir de la racine du système de fichiers (par *root*).

Dans ces conditions, les paquetages `rpm` représentent une bonne alternative.

Paquetage à la manière Red Hat

L'utilitaire **rpm** (*Red Hat Package Manager*) n'est pas du tout limité à cette distribution. Les paquetages `.rpm` sont en réalité supportés plus ou moins directement par l'essentiel des grandes distributions Linux actuelles.

Le principe de ces paquetages est d'incorporer non seulement les fichiers, mais aussi des informations sur les options de compilation, les dépendances par rapport à d'autres éléments du système (bibliothèques, utilitaires…), ainsi que la documentation des logiciels. Ces paquetages permettent naturellement d'intégrer au besoin le code source de l'application.

La création d'un paquetage nécessite un peu plus d'attention que l'utilisation de `tar`, car il faut passer par un fichier intermédiaire de spécifications. Par contre, l'utilisation au niveau de

l'administrateur qui installe le produit est très simple. Il a facilement accès à de nombreuses possibilités, en voici quelques exemples :

- Installation ou mise à jour d'un nouveau paquetage :

```
rpm -U paquet.rpm
```

- Mise à jour uniquement si une ancienne version était déjà installée :

```
$ rpm -F paquet.rpm
```

- Suppression d'un paquetage :

```
$ rpm -e paquet
```

- Recherche du paquetage contenant un fichier donné :

```
$ rpm -qf /usr/local/bin/fichier
```

- Liste de tous les paquets installés et recherche de ceux qui ont un nom donné :

```
$ rpm -qa | grep nom
```

La page de manuel de `rpm` est assez complète, et il existe de surcroît un document nommé *RPM-HOWTO* aidant à la création de paquetages.

Environnements de développement intégré

Il existe sous Linux plusieurs environnements de développement intégré permettant de gérer l'ensemble du projet à l'aide d'une seule application. Dans tous les cas, il ne s'agit que de frontaux qui sous-traitent les travaux aux outils dont nous avons parlé ci-dessus, compilateur, éditeur de liens, débogueur, etc.

L'intérêt d'un tel environnement est avant tout de gérer les dépendances entre les modules et de faciliter la compilation séparée des fichiers. Ceci simplifie le travail par rapport à la rédaction d'un fichier `Makefile` comme nous l'avons vu.

Il existe plusieurs environnements. Le plus connu est probablement **KDevelop**, issu du projet KDE.

Un autre environnement souvent employé est **Eclipse**, fourni sous licence Open Source. Parmi les nombreux environnements commerciaux, **Code Warrior** de *Metrowerks* est apparemment l'un des plus répandus.

Tous ces outils permettent de disposer d'une interface qui aidera des développeurs provenant du monde Windows à s'acclimater à la programmation sous Linux. Tous emploient aussi les outils Gnu pour les compilations, débogage, etc. Il n'y a donc pas de grosses différences de performances entre ces environnements, hormis l'ergonomie de l'interface graphique.

Contrôle de version

Il est tout à fait possible de réaliser un gros développement logiciel réunissant de nombreux programmeurs sans employer de système de contrôle automatique des versions. Le noyau Linux lui-même en est un bon exemple. Toutefois l'utilisation d'un outil comme **RCS** (*Revision Control System*) simplifie la mise au point d'une application, principalement lorsque les

modifications sont espacées dans le temps et qu'on n'a plus nécessairement en mémoire la liste des corrections apportées en dernier.

Le système *RCS* est l'équivalent de l'utilitaire **SCCS** (*Source Code Control System*) qu'on trouve sur les Unix commerciaux. Le principe consiste à verrouiller individuellement chaque fichier source d'une application. Avant de modifier un fichier, le programmeur devra demander son extraction de la base avec la commande `co` (*Check Out*) puis, après l'édition, il invoquera `ci` (*Check In*) pour rendre le fichier au système.

Naturellement *RCS* garde une trace de chaque modification apportée, avec un descriptif en clair. Il existe des commandes supplémentaires, comme `rcsmerge` pour imposer une modification sur une ancienne version, `rlog` pour examiner l'historique d'un fichier, `ident` pour rechercher des chaînes d'identification, ou `rcsdiff` qui compare deux versions d'un même fichier. On consultera leurs pages de manuel respectives pour obtenir des informations sur ces commandes, en commençant par `rcsintro`.

En fait, les limitations de *RCS* apparaissent dès que plusieurs développeurs travaillent sur le même projet. Pour éviter qu'un même fichier soit modifié simultanément par plusieurs personnes, *RCS* impose qu'une copie seulement puisse être extraite pour être modifiée. Plusieurs personnes peuvent réclamer une copie en lecture seule du même fichier, mais on ne peut extraire qu'une seule copie en lecture et écriture.

Si on désire utiliser *RCS* durant la phase de développement d'un logiciel, alors que chaque fichier est modifié plusieurs fois par jour, ce système nécessite une très forte communication au sein de l'équipe de programmeurs (en clair tout le monde doit travailler dans le même bureau).

Par contre, sa mise en œuvre est plus raisonnable une fois que le projet commence à atteindre son terme, et qu'on passe en phase de tests et de débogage. Il est alors utile de conserver une trace exacte des modifications apportées, de leurs buts et des circonstances dans lesquelles elles ont été décidées.

Lorsque le nombre de développeurs est plus important, il est possible d'employer un autre mécanisme de suivi, **CVS** (*Concurrent Version System*). Le principe est quelque peu différent. *CVS* conserve une copie centralisée de l'arborescence des sources, et chaque développeur peut disposer de sa propre copie locale. Lorsque des modifications ont été apportées à des fichiers source locaux, le programmeur peut alors publier ses changements. *CVS* assure alors une mise à jour des sources maîtresses, après avoir vérifié que les fichiers n'ont pas été modifiés entre-temps par un autre utilisateur. Lorsque des modifications sont publiées, il est recommandé de diffuser par e-mail un descriptif des changements, ce qui peut être automatisé dans la configuration de *CVS*.

La résolution des conflits se produisant si plusieurs développeurs modifient simultanément le même fichier a lieu durant la publication des modifications. Avant la publication, *CVS* impose la mise à jour de la copie locale des sources, en leur appliquant les changements qui ont pu avoir lieu sur la copie maîtresse depuis la dernière mise à jour. Ces changements sont appliqués intelligemment, à la manière de *diff* et de *patch*. Dans le pire des cas *CVS* demandera au programmeur de valider les modifications si elles se recouvrent avec ses propres corrections.

Les commandes de *CVS* sont nombreuses, mais elles se présentent toutes sous forme d'arguments en ligne de commande pour l'utilitaire `/usr/bin/cvs`. On consultera sa page de manuel pour en avoir une description complète.

L'efficacité de *CVS* est surtout appréciable durant la phase de développement de l'application et son débogage. Une fois que le logiciel est arrivé à une maturité telle qu'il n'est plus maintenu que par une seule personne qui y apporte quelques corrections de temps à autre, il est difficile de s'obliger à utiliser systématiquement les commandes de publication des modifications et de documentation des changements. Lorsqu'un projet n'est entre les mains que d'un seul développeur, celui-ci a tendance à considérer sa copie locale comme source maîtresse du système.

On peut imaginer utiliser intensivement *CVS* durant les phases actives de développement d'un logiciel, puis basculer sur *RCS* lorsqu'il n'y a plus que des corrections mineures et rares, qu'on confie à un nombre restreint de programmeurs.

Bibliothèques supplémentaires pour le développement

En fait, la bibliothèque C seule ne permet pas de construire d'application très évoluée, ou alors au prix d'un effort de codage démesuré et peu portable. Les limitations de l'interface utilisateur nous empêchent de dépasser le stade des utilitaires du type « filtre » qu'on rencontre sous Unix (`tr`, `grep`, `wc`…). Pour aller plus loin dans l'ergonomie d'une application, il est indispensable de recourir aux services de bibliothèques supplémentaires.

Celles-ci se présentent sous forme de logiciels libres, disponibles sur la majorité des systèmes Linux.

Interface utilisateur en mode texte

La première interface disponible pour améliorer l'ergonomie d'un programme en mode texte est la bibliothèque Gnu **Readline**, conçue pour faciliter la saisie de texte. Lorsqu'un programme fait appel aux routines de cette bibliothèque, l'utilisateur peut corriger facilement la ligne de saisie, en se déplaçant en arrière ou en avant, en modifiant les caractères déjà entrés, en utilisant même des possibilités de complétion du texte ou d'historique des lignes saisies.

Il est possible de configurer les touches associées à chaque action par l'intermédiaire d'un fichier d'initialisation, qui peut même accepter des directives conditionnelles en fonction du type de terminal sur lequel l'utilisateur se trouve. La bibliothèque *Readline* est par exemple employée par le shell *Bash*.

Pour l'affichage des résultats d'un programme en mode texte, il est conseillé d'employer la bibliothèque **ncurses**. Il s'agit d'un ensemble de fonctions permettant d'accéder de manière portable aux diverses fonctionnalités qu'on peut attendre d'un écran de texte, comme le positionnement du curseur, l'accès aux couleurs, les manipulations de fenêtres, de panneaux, de menus…

La bibliothèque *ncurses* disponible sous Linux est libre et compatible avec la bibliothèque *curses*, décrite par les spécifications SUSv3, présente sur l'essentiel des Unix commerciaux.

Non seulement *ncurses* nous fournit des fonctionnalités gérant tous les types de terminaux de manière transparente, mais en plus la portabilité du programme sur d'autres environnements Unix est assurée. On comprendra que de nombreuses applications y fassent appel.

Développement sous X-Window

La programmation d'applications graphiques sous X-Window peut parfois devenir un véritable défi, en fonction de la portabilité désirée pour le logiciel.

Le développement sous X-Window est organisé en couches logicielles successives. Au bas de l'ensemble se trouve la bibliothèque **Xlib**. Cette bibliothèque offre les fonctionnalités élémentaires en termes de dessin (tracé de polygones, de cercles, de texte, etc.), de fenêtrage et de récupération d'événements produits par la souris ou le clavier. La notion de fenêtrage est ici réduite à sa plus simple expression, puisqu'il s'agit uniquement de zones rectangulaires sur l'écran, sans matérialisation visible (pas de bordure).

L'appel des fonctions de la *Xlib* est indispensable dès qu'on utilise des primitives graphiques de dessin. Par contre, si on veut disposer ne serait-ce que d'un bouton à cliquer, il faut le dessiner entièrement avec ses contours, son texte, éventuellement la couleur de fond et les ombrages. Naturellement, une bibliothèque prend en charge ce travail et offre des composants graphiques élémentaires (les *widgets*).

Les fonctionnalités proposées par la couche nommée **Xt** ne sont toujours pas suffisantes, car celle-ci ne fait que définir des classes génériques d'objets graphiques et n'en offre pas d'implémentation esthétique. On peut utiliser les objets fournis par défaut avec le système X-Window dans la bibliothèque *Athena Widget*, mais ils ne sont vraiment pas très élégants (voir la figure 1.5).

Pour obtenir une bonne interface graphique, il faut donc utiliser une couche supplémentaire. Le standard le plus employé dans le domaine industriel est la bibliothèque **Motif Xm**. Assez ergonomique et plutôt agréable visuellement (voir la figure 1.4), la bibliothèque *Motif* est disponible sur tous les systèmes Unix commerciaux.

Sous Linux, le gros problème est que la licence pour les implémentations commerciales de *Motif* est relativement chère. Heureusement le projet **lesstif**, dont le développement continue activement, a produit une implémentation libre, compatible avec *Motif 1.2*. Cette bibliothèque n'est pas totalement exempte de bogues, mais elle est bien assez stable pour permettre son utilisation quotidienne par un développeur.

Il est donc possible d'écrire sous Linux des logiciels portables, au standard *Motif*, en utilisant *lesstif* pour le développement et les distributions libres, quitte à se procurer une implémentation commerciale pour l'utilisation finale, si on désire vraiment une version stable de *Motif*.

Il manque toutefois sous Linux un outil permettant de mettre facilement en place une interface graphique (en faisant glisser des boutons, des barres de défilement, etc.). Certains utilitaires existent, mais leurs performances sont généralement assez limitées. La création d'une interface graphique complète passe donc par une phase un peu rébarbative de mise en place manuelle des composants de chaque boîte de dialogue.

Les environnements Kde et Gnome

Les deux environnements homogènes les plus répandus sous Linux sont **Kde** (*K Desktop Environment*) et **Gnome** (*Gnu Network Model Environment*). L'un comme l'autre possèdent une interface de programmation très évoluée, rendant plus facile le développement de logiciels graphiques.

Ces environnements sont parfaitement appropriés pour la mise en œuvre de logiciels – libres ou commerciaux – pour Linux. Toutefois la portabilité vers d'autres Unix est sensiblement amoindrie.

L'environnement *Gnome* est construit autour de la bibliothèque graphique **GTK** (*Gimp Toolkit*), initialement développée, comme son nom l'indique, pour l'utilitaire graphique Gimp. La programmation sous *Kde* repose sur une bibliothèque nommée **Qt**, disponible gratuitement pour des développements non commerciaux, mais dont la licence est plus restrictive que celle de *GTK*. Il existe des documents sur la programmation sous *KDE* sur le Web. Pour le développement dans l'environnement *Gnome*, on consultera [ODIN 2000] *Programmation Linux avec GTK+*.

Conclusion

Ce chapitre nous aura permis de faire le point sur les outils disponibles pour le développeur dans l'environnement Linux/Gnu.

Pour avoir de plus amples informations sur l'installation et l'utilisation d'une station Linux, on consultera par exemple [WELSH 2003] *Le système Linux*.

On trouvera des conseils sur la programmation sous Unix en général dans [BOURNE 1985] *Le système Unix* et dans [RIFFLET 1995] *La programmation sous Unix*.

Enfin, les utilitaires du projet Gnu comme *emacs*, *gcc*, *gdb*, *make*… sont traités en détail dans [LOUKIDES 1997] *Programmer avec les outils Gnu*.

2

La notion de processus

Nous allons commencer notre étude de la programmation en C sous Linux par plusieurs chapitres analysant les divers aspects de l'exécution des applications. Ce chapitre introduira la notion de processus, ainsi que les différents identifiants qui y sont associés, leurs significations et leurs utilisations dans le système.

Dans les chapitres suivants, nous étudierons les interactions qu'un processus peut établir avec son environnement, c'est-à-dire sa propre vision du système, configurée par l'utilisateur, puis l'exécution et la fin des processus, en analysant toutes les méthodes permettant de démarrer un autre programme, de suivre ou de contrôler l'utilisation des ressources, de détecter et de gérer les erreurs.

Présentation des processus

Sous Unix, toute tâche qui est en cours d'exécution est représentée par un processus. Un processus est une entité comportant à la fois des données et du code. On peut considérer un processus comme une unité élémentaire en ce qui concerne l'activité sur le système.

On peut imaginer un processus comme un programme en cours d'exécution. Cette représentation est très imparfaite car une application peut non seulement utiliser plusieurs processus concurrents, mais un unique processus peut également lancer l'exécution d'un nouveau programme, en remplaçant entièrement le code et les données du programme précédent.

À un instant donné, un processus peut, comme nous le verrons plus loin, se trouver dans divers états. Le noyau du système d'exploitation est chargé de réguler l'exécution des processus afin de garantir à l'utilisateur un comportement multitâche performant. Le noyau fournit un mécanisme de régulation des tâches qu'on nomme « ordonnancement » (en anglais *scheduling*). Cela assure la répartition équitable de l'accès au microprocesseur par les divers processus concurrents.

Sur une machine uni-processeur, il n'y a qu'un seul processus qui s'exécute effectivement à un instant donné. Le noyau assure une commutation régulière entre tous les processus présents sur le système pour garantir un fonctionnement multitâche. Sur une machine multi-processeur, le principe est le même, à la différence que plusieurs processus – mais rarement tous – peuvent s'exécuter réellement en parallèle.

On peut examiner la liste des processus présents sur le système à l'aide de la commande `ps`, et plus particulièrement avec ses options `ax`, qui nous permettent de voir les processus endormis, et ceux qui appartiennent aux autres utilisateurs. On voit alors, même sur un système apparemment au repos, une bonne trentaine de processus plus ou moins actifs :

```
$ ps ax
 PID TTY  STAT  TIME COMMAND
   1 ?     S    0:03 init
   2 ?     SW   0:03 (kflushd)
   3 ?     SW<  0:00 (kswapd)
   4 ?     SW   0:00 (nfsiod)
   5 ?     SW   0:00 (nfsiod)
   6 ?     SW   0:00 (nfsiod)
   7 ?     SW   0:00 (nfsiod)
  28 ?     S    0:00 /sbin/kerneld
 194 ?     S    1:26 syslogd
 203 ?     S    0:00 klogd
 225 ?     S    0:00 crond
 236 ?     SW   0:00 (inetd)
 247 ?     SW   0:00 (lpd)
 266 ?     S    0:00 (sendmail)
 278 ?     S    0:00 gpm -t PS/2
    [...]
2045 tty4  S    0:00 /sbin/mingetty tty4
2046 tty5  S    0:00 /sbin/mingetty tty5
2047 tty6  S    0:00 /sbin/mingetty tty6
2048 ?     S    0:00 /usr/bin/gdm-binary -nodaemon
2309 ?     S    0:00 /usr/bin/gdm-binary -nodaemon
2316 ?     S    0:08 /usr/X11R6/bin/X :0 -audit 0
3109 ?     S    0:00 /bin/sh /home/ccb/.xsession
3158 ?     S    0:00 /usr/bin/ssh-agent /home/ccb/.xsession
3162 ?     S    0:00 xfwm
3167 ?     S    0:00 /usr/bin/xfce 7 4 /etc/X11/xfce/xfwmrc 0 10 XFwm
4282 ?     S    0:00 in.telnetd: 192.168.1.50
4283 ?     S    0:00 login -- ccb
4284 pts/1 S    0:00 -bash
4323 pts/1 R    0:00 ps ax
$
```

La commande `ps` affiche plusieurs colonnes dont la signification ne nous importe pas pour le moment. Retenons simplement que nous voyons en dernière colonne l'intitulé complet de la commande qui a démarré le processus, et en première colonne un numéro d'identification qu'on nomme PID.

Identification par le PID

Le premier processus du système, `init`, ainsi que quelques autres sont créés directement par le noyau au démarrage. La seule manière, ensuite, de créer un nouveau processus est d'appeler l'appel-système **`fork()`**, qui va dupliquer le processus appelant. Au retour de cet appel-système, deux processus identiques continueront d'exécuter le code à la suite de `fork()`. La différence essentielle entre ces deux processus est un numéro d'identification. On distingue ainsi le processus original, qu'on nomme traditionnellement le processus père, et la nouvelle copie, le processus fils.

L'appel-système `fork()` est déclaré dans `<unistd.h>`, ainsi :

```
pid_t fork(void);
```

Les deux processus pouvant être distingués par leur numéro d'identification PID (Process IDentifier), il est possible d'exécuter deux codes différents au retour de l'appel-système `fork()`. Par exemple, le processus fils peut demander à être remplacé par le code d'un autre programme exécutable se trouvant sur le disque. C'est exactement ce que fait un shell habituellement.

Pour connaître son propre identifiant PID, on utilise l'appel-système **`getpid()`**, qui ne prend pas d'argument et renvoie une valeur de type `pid_t`. Il s'agit, bien entendu, du PID du processus appelant. Cet appel-système, déclaré dans `<unistd.h>`, est l'un des rares qui n'échouent jamais :

```
pid_t getpid(void);
```

Ce numéro de PID est celui que nous avons vu affiché en première colonne de la commande `ps`. Sous Linux, le type `pid_t` est un entier sur 64 bits, mais ce n'est pas le cas pour tous les Unix. Pour assurer une bonne portabilité lors de l'affichage d'un PID, nous utiliserons la conversion `%ld` de `printf()`, et nous ferons explicitement une conversion de type en `long int` ainsi :

```
fprintf(stdout, "Mon PID est : %ld\n", (long) getpid());
```

La distinction entre processus père et fils peut se faire directement au retour de l'appel `fork()`. Celui-ci, en effet, renvoie une valeur de type `pid_t`, qui vaut zéro si on se trouve dans le processus fils, est négative en cas d'erreur, et correspond au PID du fils si on se trouve dans le processus père.

Voici en effet un point important : dans la plupart des applications courantes, la création d'un processus fils a pour but de faire dialoguer deux parties indépendantes du programme (à l'aide de signaux, de tubes, de mémoire partagée…). Le processus fils peut aisément accéder au PID de son père (noté PPID pour Parent PID) grâce à l'appel-système **`getppid()`**, déclaré dans `<unistd.h>` :

```
pid_t getppid(void);
```

Cette routine se comporte comme `getpid()`, mais renvoie le PID du père du processus appelant. En revanche, le processus père ne peut connaître le numéro du nouveau processus créé qu'au moment du retour du `fork()`.

> En réalité, un processus pourrait établir la liste de ses fils en analysant le `PPID` de tous les processus en cours d'exécution, par exemple à l'aide du pseudo-système de fichiers `/proc`, mais il est quand même beaucoup plus simple de mémoriser la valeur de retour de `fork()`.

On peut examiner la hiérarchie des processus en cours sur le système avec le champ PPID de la commande `ps axj` :

```
$ ps axj
PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     1     0     0 ?           -1 S        0   0:06 init [5]
    1     2     0     0 ?           -1 SWN      0   0:00 [ksoftirqd/0]
    1     3     0     0 ?           -1 SW<      0   0:00 [events/0]
    3     4     0     0 ?           -1 SW<      0   0:00 [kblockd/0]
    [...]
    1  2046  2046  2046 tty5      2046 S        0   0:00 /sbin/mingetty
    1  2047  2047  2047 tty6      2047 S        0   0:00 /sbin/mingetty
    1  2048  2048  2048 ?           -1 S        0   0:00 /usr/bin/gdm-b
 2048  2309  2309  2048 ?           -1 S        0   0:00 /usr/bin/gdm-b
 2309  2316  2316  2048 ?           -1 S        0   0:08 /usr/X11R6/bin
 2309  3109  3109  3109 ?           -1 S      500   0:00 .xsession
 3109  3158  3158  3158 ?           -1 S      500   0:00 /usr/bin/ssh-a
 3109  3162  3109  3109 ?           -1 S      500   0:00 xfwm
 3162  3167  3109  3109 ?           -1 S      500   0:00 /usr/bin/xfce
 3167  4301  3109  3109 ?           -1  S       0   0:01 xterm -ls
```

On voit que `init` n'a pas de père (PPID = 0), mais qu'un grand nombre de processus héritent de lui. On peut observer également une filiation directe `gdm` (2309) – `.xsession` (3109) – `xfwm` (3162) – `xfce` (3167) – `xterm` (4301)…

Lorsqu'un processus est créé par `fork()`, il dispose d'une *copie* des données de son père, mais également de l'environnement de celui-ci et d'un certain nombre d'autres éléments (table des descripteurs de fichiers, etc.). On parle alors d'héritage du père.

Notons que, sous Linux, l'appel-système `fork()` est très économe car il utilise une méthode de « copie sur écriture ». Cela signifie que toutes les données qui doivent être dupliquées pour chaque processus (descripteurs de fichier, mémoire allouée…) ne seront pas immédiatement recopiées. Tant qu'aucun des deux processus n'a modifié des informations dans ces pages mémoire, il n'y en a qu'un seul exemplaire sur le système. Par contre, dès que l'un des processus réalise une écriture dans la zone concernée, le noyau assure la véritable duplication des données. Une création de processus par `fork()` n'a donc qu'un coût très faible en termes de ressources système.

En cas d'erreur, `fork()` renvoie la valeur –1, et la variable globale `errno` contient le code d'erreur, défini dans `<errno.h>`. Ce code d'erreur peut être soit `ENOMEM`, qui indique que le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus, soit `EAGAIN`, `qui` signale que le système n'a plus de place libre dans sa table des processus, mais qu'il y en aura probablement sous peu. Un processus est donc autorisé à réitérer sa demande de duplication lorsqu'il a obtenu un code d'erreur `EAGAIN`.

Voici à présent un exemple de création d'un processus fils par l'appel-système `fork()`.

exemple_fork.c :

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>
```

```
    int
main (void)
{
    pid_t pid_fils;

    do {
        pid_fils = fork();
    } while ((pid_fils == -1) && (errno == EAGAIN));
    if (pid_fils == -1) {
        fprintf(stderr, "fork () impossible, errno=%d\n", errno);
        return 1;
    }
    if (pid_fils == 0) { /* processus fils */
        fprintf(stdout, "Fils : PID=%ld, PPID=%ld\n",
                (long) getpid(), (long) getppid());
        return 0;
    } else { /* processus père */
        fprintf(stdout, "Pere : PID=%d, PPID=%d, PID fils=%ld\n",
                (long) getpid(),(long) getppid(),(long)pid_fils);
        wait(NULL);
        return 0;
    }
}
```

Lors de son exécution, ce programme fournit les informations suivantes :

```
$ echo $$
4284
$ ./exemple_fork
Pere : PID=4340, PPID=4284, PID fils=4341
Fils : PID=4341, PPID=4340
$
```

La variable spéciale `$$` du shell correspond à son PID. Ceci nous permet de voir que le PPID du processus père correspond au shell.

Dans notre exemple, l'appel-système `fork()` boucle si le noyau n'a plus assez de place dans sa table interne pour créer un nouveau processus. Dans ce cas, le système est déjà probablement dans une situation assez critique, et il n'est pas utile de gâcher des ressources CPU en effectuant une boucle hystérique sur `fork()`. Il serait préférable d'introduire un délai d'attente dans notre code pour ne pas réitérer notre demande immédiatement, et attendre ainsi pendant quelques secondes que le système revienne dans un état plus calme. Nous verrons des moyens d'endormir le processus dans le chapitre 9.

On remarquera que nous avons introduit un appel-système `wait(NULL)` à la fin du code du père. Nous en reparlerons ultérieurement, mais on peut d'ores et déjà noter que cela permet d'attendre la fin de l'exécution du fils. Si nous n'avions pas employé cet appel-système, le processus père aurait pu se terminer avant son fils, redonnant la main au shell, qui aurait alors affiché son symbole d'invite (`$`) avant que le fils n'ait imprimé ses informations. Voici ce qu'on aurait pu observer :

```
$ ./exemple_fork
Pere : PID=4344, PPID=4284, PID fils=4345
$ Fils : PID=4345, PPID=4344
```

Identification de l'utilisateur correspondant au processus

À l'opposé des systèmes mono-utilisateurs, un système Unix est particulièrement orienté vers l'identification de ses utilisateurs. Toute activité entreprise par un utilisateur est soumise à des contrôles stricts quant aux permissions qui lui sont attribuées. Pour cela, chaque processus s'exécute sous une identité précise. Dans la plupart des cas, il s'agit de l'identité de l'utilisateur qui a invoqué le processus et qui est définie par une valeur numérique : l'UID (User IDentifier). Dans certaines situations que nous examinerons plus bas, il est nécessaire pour le processus de changer d'identité.

La commande `shell id` affiche les identifiants et les groupes auxquels appartient l'utilisateur.

Il existe trois identifiants d'utilisateur par processus : l'UID réel, l'UID effectif, et l'UID sauvé. L'UID réel est celui de l'utilisateur ayant lancé le programme. L'UID effectif est celui qui correspond aux privilèges accordés au processus. L'UID sauvé est une copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus.

L'essentiel des ressources sous Unix (données, périphériques…) s'exprime sous forme de nœuds du système de fichiers. Lors d'une tentative d'accès à un fichier, le noyau effectue des vérifications d'autorisation en prenant en compte l'UID effectif du processus appelant. Généralement, cet UID effectif est le même que l'UID réel (celui de la personne ayant invoqué le processus). C'est le cas de toutes les applications classiques ne nécessitant pas de privilège particulier, par exemple les commandes Unix classiques (`ls`, `cp`, `mv`…) qui s'exécutent sous l'identité de leur utilisateur, laissant au noyau le soin de vérifier les permissions d'accès.

Certaines applications peuvent toutefois avoir besoin – souvent ponctuellement – d'autorisations spéciales, tout en étant invoquées par n'importe quel utilisateur. L'exemple le plus évident est `su`, qui permet de changer d'identité, mais on peut en citer beaucoup d'autres, comme `mount`, qui peut autoriser sous Linux tout utilisateur à monter des systèmes de fichiers provenant d'un CD-Rom ou d'une disquette, par exemple. Il y a également les applications utilisant des couches basses des protocoles réseau comme `ping`. Dans ce cas, il faut que le processus garde son UID réel pour savoir qui agit, mais il dispose d'un UID effectif lui garantissant une liberté suffisante sur le système pour accéder aux ressources désirées.

Les appels-système **`getuid()`** et **`geteuid()`** permettent respectivement d'obtenir l'UID réel et l'UID effectif du processus appelant. Ils sont déclarés dans `<unistd.h>`, ainsi :

```
uid_t getuid  (void);
uid_t geteuid (void);
```

Le type `uid_t` correspondant au retour des fonctions `getuid()` et `geteuid()` est défini dans `<sys/types.h>`. Il s'agit d'un entier non signé. Depuis Linux 2.4, ce type occupe 32 bits ; auparavant il tenait sur 16 bits. Nous utiliserons la conversion `%u` pour `printf()` pour garantir une bonne portabilité.

L'UID effectif est différent de l'UID réel lorsque le fichier exécutable dispose d'un attribut particulier permettant au processus de changer d'identité au démarrage du programme. Considérons par exemple le programme suivant.

exemple_getuid.c :

```
#include <stdio.h>
#include <unistd.h>
```

```
    int
main(void)
{
    fprintf(stdout, "  UID réel = %u, UID effectif = %u\n",
         getuid(), geteuid());
    return 0;
}
```

Quand on compile ce programme, on obtient un fichier exécutable, qu'on lance ensuite :

```
$ ls -ln exemple_getuid*
  rwxr-xr-x   1 500      500      4446 Jun 10 10:56 exemple_getuid
  rw-r--r--   1 500      500       208 Jun 10 10:56 exemple_getuid.c
$ ./exemple_getuid UID réel = 500, UID effectif = 500
$
```

Le comportement est pour l'instant parfaitement normal. Imaginons maintenant que root passe par là, s'attribue le fichier exécutable et lui ajoute le bit « Set-UID » à l'aide de la commande `chmod`. Lorsqu'un utilisateur va maintenant exécuter `exemple_getuid`, le système va lui fournir l'UID effectif du propriétaire du fichier, à savoir root (qui a toujours l'UID 0 par définition) :

```
$ su
Password:
# chown root.root exemple_getuid
# chmod u+s exemple_getuid
# ls -ln exemple_getuid*
  rwsr-xr-x   1 0        0        4446 Jun 10 10:56 exemple_getuid
  rw-r--r--   1 500      500       208 Jun 10 10:56 exemple_getuid.c
# ./exemple_getuid
UID réel = 0, UID effectif = 0
# exit
$ ./exemple_getuid
UID réel = 500, UID effectif = 0
$
```

Nous voyons l'attribut Set-UID indiqué par la lettre « `s` » dans les autorisations d'accès. L'UID réel est conservé à des fins d'identification éventuelle au sein du processus.

Notre programme ayant l'UID effectif de root en a tous les privilèges. Vous pouvez en avoir le cœur net en lui faisant, par exemple, créer un nouveau fichier dans le répertoire `/etc`. Si vous n'avez pas les privilèges root sur votre système, vous pouvez néanmoins effectuer les tests en accord avec un autre utilisateur qui copiera votre exécutable dans son répertoire personnel (pour en prendre possession) et lui ajoutera le bit Set-UID.

Il existe plusieurs appels-système permettant à un processus de modifier son UID. Il ne peut toutefois s'agir que de perdre des privilèges, éventuellement d'en retrouver des anciens, mais jamais d'en gagner. Imaginons un émulateur de terminal série (un peu comme `kermit` ou `minicom`). Il a besoin d'accéder à un périphérique système (le modem), même en étant lancé par n'importe quel utilisateur. Il dispose donc de son bit Set-UID activé, tout en appartenant à root. Cela lui permet d'ouvrir le fichier spécial correspondant au périphérique et de gérer la liaison.

Toutefois, il faut également sauvegarder sur disque des informations n'appartenant qu'à l'utilisateur ayant lancé l'application (sa configuration préférée pour l'interface, par exemple), voire enregistrer dans un fichier un historique complet de la session. Pour ce faire, le programme ne doit créer des fichiers que dans des endroits où l'utilisateur est autorisé à le faire. Plutôt que de vérifier toutes les autorisations d'accès, il est plus simple de perdre temporairement ses privilèges root pour reprendre l'identité de l'utilisateur original, le temps de faire l'enregistrement (les permissions étant alors vérifiées par le noyau), et de redevenir éventuellement root ensuite. Nous reviendrons à plusieurs reprises sur ce mécanisme.

Le troisième type d'UID d'un processus est l'UID sauvé. Il s'agit d'une copie de l'ancien UID effectif lorsque celui-ci est modifié par l'un des appels décrits ci-dessous. Cette copie est effectuée automatiquement par le noyau. Un processus peut toujours demander à changer son UID effectif ou son UID réel pour prendre la valeur de l'UID sauvé. Il est également possible de prendre en UID effectif la valeur de l'UID réel, et inversement.

Un processus avec le bit Set-UID positionné démarre donc avec un UID effectif différent de celui de l'utilisateur qui l'a invoqué. Quand il désire effectuer une opération non privilégiée, il peut demander à remplacer son UID effectif par l'UID réel. Une copie de l'UID effectif est conservée dans l'UID sauvé. Il pourra donc à tout moment demander à remplacer à nouveau son UID effectif par son UID sauvé.

Pour cela, il existe – pour des raisons historiques – plusieurs appels-système permettant sous Linux de modifier son UID : **`setuid()`**, **`seteuid()`**et **`setreuid()`** sont définis par SUSv3 ; **`setresuid()`**est spécifique à Linux.

Les trois premiers appels-système sont déclarés dans `<unistd.h>`, ainsi :

```
int setuid   (uid_t uid_effectif);
int seteuid  (uid_t uid_effectif);
int setreuid (uid_t uid_reel, uid_t uid_effectif);
```

Ils permettent de modifier un ou plusieurs UID du processus appelant, renvoyant 0 s'ils réussissent, ou –1 en cas d'échec.

Nous allons voir le comportement d'un programme Set-UID qui abandonne temporairement ses privilèges pour disposer des permissions de l'utilisateur l'ayant invoqué, puis qui reprend à nouveau ses autorisations originales. Notez bien que, dans cette première version, la récupération de l'ancienne identité *ne fonctionne pas* si le programme appartient à root. Ceci est clairement défini dans l'implémentation de `setuid()`. Les développeurs de Linux préviennent bien qu'en cas de mécontentement, il faut s'en prendre au comité Posix, qui est responsable de cette règle. Nous verrons immédiatement après une version utilisant `setreuid()`, qui fonctionne dans tous les cas de figure.

exemple_setuid.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (void)
{
    uid_t       uid_reel;
    uid_t       uid_eff;
```

```
    uid_reel = getuid();
    uid_eff = geteuid();
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());
    fprintf(stdout, " setuid(%d) = %d\n", uid_reel, setuid(uid_reel));
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());
    fprintf(stdout, " setuid(%d) = %d\n", uid_eff, setuid(uid_eff));
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());

    return 0;
}
```

L'exécution du programme (copié par un autre utilisateur, et avec le bit Set-UID positionné) donne :

```
$ ls -ln exemple_setuid*
-rwsr-xr-x   1 501      501      4717 Jun 10 15:49 exemple_setuid
$ ./exemple_setuid
  UID réel = 500,  UID effectif = 501
 setuid(500) = 0
  UID réel = 500,  UID effectif = 500
 setuid(501) = 0
  UID réel = 500,  UID effectif = 501
$
```

Si on tente la même opération avec un programme Set-UID root, il ne pourra plus reprendre ses privilèges, car lorsque `setuid()` est invoqué par un utilisateur ayant un UID effectif nul (root), il écrase également l'UID sauvé pour empêcher le retour en arrière.

Voici maintenant une variante utilisant l'appel-système `setreuid()`. Comme on peut s'en douter, il permet de fixer les deux UID en une seule fois. Si l'un des deux UID vaut –1, il n'est pas changé. Cet appel-système à longtemps été réservé aux Unix BSD, mais il est dorénavant défini par SUSv3 et ne devrait pas poser de problèmes de portabilité sur les systèmes récents.

exemple_setreuid.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (void)
{
    uid_t       uid_reel;
    uid_t       uid_eff;

    uid_reel = getuid();
    uid_eff = geteuid();
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());
    fprintf(stdout, " setreuid(-1, %d) = %d\n", uid_reel,
                        setreuid(-1, uid_reel));
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());
    fprintf(stdout, " setreuid(-1, %d) = %d\n", uid_eff,
                        setreuid(-1, uid_eff));
```

```
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());
    fprintf(stdout, " setreuid(%d, -1) = %d\n", uid_eff,
                         setreuid(uid_eff, -1));
    fprintf(stdout, " UID-R = %u,  UID-E = %u\n", getuid(), geteuid());

    return 0;
}
```

En voici l'exécution, après passage en Set-UID root :

```
$ ls -ln exemple_setre*
  rwsrwsr-x   1 0        0        4809 Jun 10 16:23 exemple_setreuid
  rw-rw-r--   1 500      500       829 Jun 10 16:23 exemple_setreuid.c
$ ./exemple_setreuid
 UID-R = 500,  UID-E = 0
setreuid(-1, 500) = 0
 UID-R = 500,  UID-E = 500
setreuid(-1, 0) = 0
UID-R = 500,  UID-E = 0
setreuid(0, -1) = 0
UID-R = 0,  UID-E = 0
$
```

Cette fois-ci, le changement fonctionne parfaitement, même avec un UID effectif nul.

Enfin, il est possible – mais c'est une option spécifique à Linux – de modifier également l'UID sauvé, principalement pour empêcher le retour en arrière comme le fait `setuid()`, avec l'appel-système **`setresuid()`**. En raison de la spécificité de cet appel-système, il faut définir la constante `_GNU_SOURCE` avant d'inclure le fichier d'en-tête `<unistd.h>`.

```
int setresuid (uid_t uid_reel, uid_t uid_effectif, uid_t uid_sauve);
int getresuid (uid_t * uid_reel, uid_t * uid_effectif, uid_t * uid_sauve);
```

exemple_setresuid.c :

```
    #define _GNU_SOURCE

    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (void)
{
    uid_t  uid_R, uid_E, uid_S;

    getresuid(& uid_R, & uid_E, & uid_S);
    printf("UID-R=%u, UID-E=%u, UID-S=%u\n", uid_R, uid_E,uid_S);
    printf(" setresuid(-1, %u, %u)=%d\n",
                uid_E, uid_R, setresuid(-1, uid_E, uid_R));
    getresuid(& uid_R, & uid_E, & uid_S);
    printf("UID-R=%u, UID-E=%u, UID-S=%u\n", uid_R, uid_E,uid_S);
    printf(" setresuid(-1, %u, -1)=%d\n",
                uid_S, setresuid(-1, uid_S, -1));
    getresuid(& uid_R, & uid_E, & uid_S);
```

```
    printf("UID-R=%u, UID-E=%u, UID-S=%u\n", uid_R, uid_E,uid_S);
    return 0;
}
```

L'exécution est intéressante si le programme est installé Set-UID *root* :

```
$ ls -ln exemple_setresuid
-rwsr-xr-x   1 0        0       12404 Nov 14 15:10 exemple_setresuid
$ ./exemple_setresuid
UID-R=500, UID-E=0, UID-S=0
 setresuid(-1, 500, 0)=0
UID-R=500, UID-E=500, UID-S=0
 setresuid(-1, 0, -1)=0
UID-R=500, UID-E=0, UID-S=0
$
```

Identification du groupe d'utilisateurs du processus

Chaque utilisateur du système appartient à un ou plusieurs groupes. Ces derniers sont définis dans le fichier `/etc/groups`. Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé. Comme nous l'avons vu avec les UID, un processus dispose donc de plusieurs GID (Group IDentifier) réel, effectif, sauvé, ainsi que de GID supplémentaires si l'utilisateur qui a lancé le processus appartient à plusieurs groupes.

> **Attention**
> Il ne faut pas confondre les *groupes d'utilisateurs* auxquels un processus appartient, et qui dépendent de la personne qui lance le processus et éventuellement des attributs Set-GID du fichier exécutable, avec les *groupes de processus*, qui permettent principalement d'envoyer des signaux à des ensembles de processus. Un processus appartient donc à deux types de groupes qui n'ont rien à voir les uns avec les autres.

Le GID réel correspond au groupe principal de l'utilisateur ayant lancé le programme (celui qui est mentionné dans `/etc/passwd`).

Le GID effectif peut être différent du GID réel si le fichier exécutable dispose de l'attribut Set-GID (`chmod g+s`). C'est le GID effectif qui est utilisé par le noyau pour vérifier les autorisations d'accès aux fichiers.

La lecture de ces GID se fait symétriquement à celle des UID avec les appels-système **`getgid()`** et **`getegid()`**. La modification (sous réserve d'avoir les autorisations nécessaires) peut se faire à l'aide des appels **`setgid()`**, **`setegid()`** et **`setregid()`**. Les prototypes de ces fonctions sont présents dans `<unistd.h>`, le type `gid_t` étant défini dans `<sys/types.h>` :

```
gid_t getgid(void);
gid_t getegid(void);
int setgid(gid_t egid);
int setegid(gid_t egid);
int setregid(gid_t rgid, gid_t egid);
```

Les deux premières fonctions renvoient le GID demandé, les deux dernières renvoient 0 si elle réussissent et –1 en cas d'échec.

L'ensemble complet des groupes auxquels appartient un utilisateur est indiqué dans `/etc/groups` (en fait, c'est une table inversée puisqu'on y trouve la liste des utilisateurs appartenant

à chaque groupe). Un processus peut obtenir cette liste en utilisant l'appel-système **getgroups()** :

```
int getgroups (int taille, gid_t liste []);
```

Celui-ci prend deux arguments, une dimension et une table. Le premier argument indique la taille (en nombre d'entrées) de la table fournie en second argument. L'appel-système va remplir le tableau avec la liste des GID supplémentaires du processus. Si le tableau est trop petit, `getgroups()` échoue (renvoie –1 et remplit `errno`), sauf si la taille est nulle ; auquel cas, il renvoie le nombre de groupes supplémentaires du processus. La manière correcte d'utiliser `getgroups()` est donc la suivante.

exemple_getgroups.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    int     taille;
    gid_t * table_gid = NULL;
    int     i;

    if ((taille = getgroups(0, NULL)) < 0) {
        fprintf(stderr, "Erreur getgroups, errno = %d\n", errno);
        return 1;
    }

    if ((table_gid = calloc(taille, sizeof(gid_t))) == NULL) {
        fprintf(stderr, "Erreur  calloc, errno = %d\n", errno);
        return 1;
    }

    if (getgroups(taille, table_gid) < 0) {
        fprintf(stderr, "Erreur getgroups, errno = %d\n", errno);
        return 1;
    }
    for (i = 0; i < taille; i ++)
        fprintf(stdout, "%u ", table_gid[i]);
    fprintf(stdout, "\n");

    free(table_gid);
    return 0;
}
```

qui donne :

```
$ ./exemple_getgroups
500 100
$
```

Le nombre maximal de groupes auxquels un utilisateur peut appartenir est défini dans `<asm/param.h>` sous le nom **`NGROUPS`**. Cette constante symbolique vaut 32 par défaut sous Linux.

Il est possible de fixer sa liste de groupes supplémentaires. La fonction **`setgroups()`** n'est néanmoins utilisable que par root (ou un processus dont le fichier exécutable est Set-UID root)[1]. Contrairement à `getgroups()`, le prototype est inclus dans le fichier `<grp.h>` de la bibliothèque GlibC 2 :

```
int setgroups (size_t taille, const gid_t * table);
```

Il faut définir la constante symbolique `_BSD_SOURCE` pour avoir accès à cette fonction.

exemple_setgroups.c :

```
    #define _BSD_SOURCE

    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <errno.h>
    #include <grp.h>

    int
main (int argc, char * argv [])
{
    gid_t * table_gid = NULL;
    int     i;
    int     taille;

    if (argc < 2) {
        fprintf(stderr, "Usage %s GID ...\n", argv[0]);
        return 1;
    }
    if ((table_gid = calloc(argc - 1, sizeof(gid_t))) == NULL) {
        fprintf(stderr, "Erreur calloc, errno = %d\n", errno);
        return 1;
    }
    for (i = 1; i < argc ; i ++)
        if (sscanf(argv[i], "%u", & (table_gid[i - 1])) != 1) {
            fprintf(stderr, "GID invalide : %s\n", argv[i]);
            return 1;
        }
    if (setgroups(i - 1, table_gid) < 0) {
        fprintf(stderr, "Erreur setgroups, errno = %d\n", errno);
        return 1;
    }
    free(table_gid);

    /* Passons maintenant à la vérification des groupes */
    fprintf(stdout, "Vérification : ");
```

1. En réalité, depuis Linux 2.2, il suffit que le processus ait la capacité `CAP_SETGID` comme nous le verrons en fin de chapitre.

```
    /*                ...
     * même code que la fonction main() de exemple_getgroups.c
     *                ...
     */
}
```

Ce programme ne fonctionne que s'il est Set-UID root :

```
$ ls -ln exemple_setgroups*
-rwxrwxr-x   1 500      500          5606 Jun 11 14:10 exemple_setgroups
-rw-rw-r--   1 500      500          1612 Jun 11 14:05 exemple_setgroups.c
$ ./exemple_setgroups 501 502 503
Erreur setgroups, errno = 1
$ su
Password:
# chown root.root exemple_setgroups
# chmod +s exemple_setgroups
# exit
$ ls -ln exemple_setgroups*
-rwsrwsr-x   1 0        0            5606 Jun 11 14:10 exemple_setgroups
-rw-rw-r--   1 500      500          1612 Jun 11 14:05 exemple_setgroups.c
$ ./exemple_setgroups 501 502 503
Vérification : 501 502 503
$
```

Pour un processus Set-UID root, le principal intérêt de la modification de la liste des groupes auxquels appartient un processus est de pouvoir ajouter un groupe spécial (donnant par exemple un droit de lecture et d'écriture sur un fichier spécial de périphérique) à sa liste, et de changer ensuite son UID effectif pour continuer à s'exécuter sous l'identité de l'utilisateur, tout en gardant le droit d'agir sur ledit périphérique.

Tout comme nous l'avons vu plus haut avec les UID, il existe sous Linux un GID sauvé pour chaque processus. Cela permet de modifier son GID effectif (en reprenant temporairement l'identité réelle), puis de retrouver le GID effectif original (qui était probablement fourni par le bit Set-GID). Pour accéder aux GID sauvés, deux appels-système, **setresgid()** et **getresgid(),** sont disponibles lorsque la constante `_GNU_SOURCE` est définie avant d'inclure `<unistd.h>` :

```
int setresgid (gid_t uid_reel, uid_t uid_effectif, uid_t uid_sauve);
int getresgid (gid_t * uid_reel, * uid_t uid_effectif, * uid_t uid_sauve);
```

Le programme `exemple_setresgid.c` est une copie de `exemple_setresuid.c` dans lequel on a changé toutes les occurrences de `uid` en `gid`. En voici un exemple d'exécution après sa transformation en programme Set-GID root :

```
$ ls -ln ./exemple_setresgid
-rwxrwsr-x   1 0        0     12404 Nov 14 15:38 ./exemple_setresgid
$ ./exemple_setresgid
  GID-R = 500,  GID-E = 0, GID-S = 0
 setresgid (-1, 500, 0) = 0
  GID-R = 500,  GID-E = 500, GID-S = 0
 setresgid (-1, 0, -1) = 0
  GID-R = 500,  GID-E = 0, GID-S = 0
$
```

Identification du groupe de processus

Les processus sont organisés en groupes. Rappelons qu'il ne faut pas confondre les groupes de processus avec les groupes d'utilisateurs que nous venons de voir, auxquels appartiennent les processus. Les groupes de processus permettent l'envoi global de signaux à un ensemble de processus. Ce concept, tout comme l'identificateur de session que nous verrons immédiatement à la suite, sert surtout aux interpréteurs de commandes – les shells – pour implémenter le contrôle des jobs. La prise en considération des groupes de processus dans les applications classiques est rare.

Pour savoir à quel groupe appartient un processus donné, on utilise l'appel-système **`getpgid()`**, déclaré dans `<unistd.h>` :

```
pid_t   getpgid (pid_t pid);
```

Celui-ci prend en argument le PID du processus visé et renvoie son numéro de groupe, ou –1 si le processus mentionné n'existe pas. Avec la bibliothèque GlibC, `getpgid()` n'est défini dans `<unistd.h>` que si la constante symbolique `_GNU_SOURCE` est déclarée avant l'inclusion.

exemple_getpgid.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv[])
{
    int     i;
    long int pid;
    long int pgid;

    if (argc == 1) {
        fprintf(stdout, "%d : %d\n", getpid(), getpgid(0));
        return 0;
    }
    for (i = 1; i < argc; i ++)
        if (sscanf(argv[i], "%ld", & pid) != 1) {
            fprintf(stderr, "PID invalide : %s\n", argv[i]);
        } else {
            pgid = (long) getpgid((pid_t) pid);
            if (pgid == -1)
                fprintf(stderr, "%ld inexistant\n", pid);
            else
                fprintf(stderr, "%ld : %d\n", pid, pgid);
        }
    return 0;
}
```

Ce programme permet de consulter les groupes de n'importe quels processus, « 0 » signifiant « processus appelant ».

```
$ ps
 PID TTY STAT TIME COMMAND
4519  p1 S  0:00 -bash
4565  p0 S  0:00 -bash
5017  p1 S  0:00 man getpgid
5018  p1 S  0:00 sh -c (cd /usr/man/fr_FR ; /usr/bin/gtbl /usr/man/fr_FR/m
5019  p1 S  0:00 sh -c (cd /usr/man/fr_FR ; /usr/bin/gtbl /usr/man/fr_FR/m
5022  p1 S  0:00 /usr/bin/less -is
5026  p0 R  0:00 ps
$ ./exemple_getpgid 4519 4565 5017 5018 5019 5022 5026 0
4519 : 4519
4565 : 4565
5017 : 5017
5018 : 5017
5019 : 5017
5022 : 5017
5026 inexistant
0 : 5027
$
```

Un groupe a été créé au lancement du processus 5017 (man), et il comprend tous les descendants (mise en forme et affichage de la page). Le processus dont le PID est identique au numéro de groupe est nommé leader du groupe. Un groupe n'a pas nécessairement de leader, celui-ci pouvant se terminer alors que ses descendants continuent de s'exécuter.

Il existe un appel-système **`getpgrp()`**, qui ne prend pas d'argument et renvoie le numéro de groupe du processus appelant, exactement comme `getpgid(0)`. Attention toutefois, la portabilité de cet appel-système n'est pas assurée, certaines versions d'Unix l'implémentant comme un synonyme exact de `getpgid()`.

exemple_getpgrp.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv[])
{
    fprintf(stdout, "%ld : %ld\n", (long) getpid(), (long) getpgrp());
    return 0;
}

$ ./exemple_getpgrp
7344 : 7344
$
```

La plupart des applications n'ont pas à se préoccuper de leur groupe de processus, mais cela peut parfois être indispensable lorsqu'on désire envoyer un signal à tous les descendants d'un processus père. Les interpréteurs de commandes, ou les programmes qui lancent des applications diverses (gestionnaires de fenêtres X11, gestionnaires de fichiers…), doivent pouvoir tuer tous les descendants directs d'un processus fils. Cela peut aussi être nécessaire si

l'application crée de nombreux processus fils (par exemple à chaque demande de connexion pour un démon serveur réseau) et désire pouvoir se terminer complètement en une seule fois.

Un processus peut modifier son propre identifiant de groupe ou celui de l'un de ses descendants grâce à l'appel-système **setpgid()** :

```
int setpgid (pid_t pid, pid_t pgid);
```

Le premier argument correspond au PID du processus à modifier. Si cet argument est nul, on considère qu'il s'agit du processus appelant. Le second argument indique le nouveau numéro de groupe pour le processus concerné. Si le second argument est égal au premier ou s'il est nul, le processus devient leader de son groupe.

L'appel-système échoue si le processus visé n'est ni le processus appelant ni l'un de ses descendants. Par ailleurs, un processus ne peut plus modifier le groupe de l'un de ses descendants si celui-ci a effectué un appel à l'une des fonctions de la famille `exec()`. Généralement, les interpréteurs de commandes utilisent la procédure suivante :

- Le shell exécute un `fork()`. Le processus père en garde le résultat dans une variable pid_fils.
- Le processus fils demande à devenir leader de son groupe en invoquant `setpgid(0,0)`.
- De manière redondante, le processus père réclame que son fils devienne leader de son groupe, cela pour éviter tout problème de concurrence d'exécution. Le père exécute donc `setpgid (pid_fils, pid_fils)`.
- Le père peut alors attendre, par exemple, la fin de l'exécution du fils avec `waitpid()`.
- Le fils appelle une fonction de la famille `exec()` pour lancer la commande désirée.

Le shell pourra alors contrôler l'ensemble des processus appartenant au groupe du fils en leur envoyant des signaux (`STOP`, `CONT`, `TERM`...). Le double appel à `setpgid()` dans le père et le fils est nécessaire car nous devons être sûrs que la modification est réalisée avant que le fils n'appelle `exec()` (donc l'invocation dans le processus père seul n'est pas suffisante) et avant que le père ne commence à lui envoyer des signaux (symétriquement l'appel dans le processus fils n'est pas suffisant).

Il existe un appel-système **setpgrp(),** qui sert directement à créer un groupe de processus et à en devenir leader. Il s'agit d'un synonyme de `setpgid(0, 0)`. Attention là encore à la portabilité de cet appel-système, car sous BSD il s'agit d'un synonyme de `setpgid()` utilisant donc deux arguments.

Identification de session

Il existe finalement un dernier regroupement de processus, les sessions, qui réunissent divers groupes de processus. Les sessions sont très liées à la notion de terminal de contrôledes processus. Il n'y a guère que les shells ou les gestionnaires de fenêtres pour les environnements graphiques qui ont besoin de gérer les sessions. Une exception toutefois : les applications qui s'exécutent sous forme de démon doivent accomplir quelques formalités concernant leur session. C'est donc principalement ce point de vue qui nous importera ici.

Généralement, une session est attachée à un terminal de contrôle, celui qui a servi à la connexion de l'utilisateur. Avec l'évolution des systèmes, les terminaux de contrôle sont souvent des pseudo-terminaux virtuels gérés par les systèmes graphiques de fenêtrage ou par

les pilotes de connexion réseau, comme nous le verrons dans le chapitre 33. Au sein d'une session, un groupe de processus est en avant-plan ; il reçoit directement les données saisies sur le clavier du terminal, et peut afficher ses informations de sortie sur l'écran de celui-ci. Les autres groupes de processus de la session s'exécutent en arrière-plan. Leur interaction avec le terminal sera étudiée ultérieurement dans le chapitre sur les signaux.

Pour créer une nouvelle session, un processus ne doit pas être leader de son groupe. En effet, la création de la session passe par une étape de constitution d'un nouveau groupe de processus prenant l'identifiant du processus appelant. Il est indispensable que cet identifiant ne soit pas encore attribué à un groupe qui pourrait contenir éventuellement d'autres processus.

La création d'une session s'effectue par l'appel-système **setsid()**, déclaré dans <unistd.h> :

```
pid_t setsid (void);
```

Il renvoie le nouvel identifiant de session, de type pid_t. Lors de cet appel, un nouveau groupe est créé, il ne contient que le processus appelant (qui en est donc le leader). Puis, une nouvelle session est créée, ne contenant pour le moment que ce groupe. Cette session ne dispose pas de terminal de contrôle. Elle devra en récupérer un explicitement si elle le désire. Les descendants du processus leader se trouveront, bien entendu, dans cette nouvelle session.

Un point de détail reste à préciser. Pour être sûr que le processus initial n'est pas leader de son groupe, on utilise généralement l'astuce suivante :

- Un processus père exécute un fork(), suivi d'un exit().
- Le processus fils se trouvant dans le même groupe que son père ne risque pas d'être leader, et peut donc tranquillement invoquer setsid().

La fonction **getsid()** prend en argument un PID et renvoie l'identifiant de la session, c'est-à-dire le PID du processus leader :

```
pid_t getsid (pid_t pid);
```

Cet appel-système n'est déclaré dans <unistd.h> que si la constante _GNU_SOURCE est définie avant son inclusion. Cette fonction n'échoue que si le PID transmis ne correspond à aucun processus existant. Comme d'habitude, getsid(0) renvoie l'identifiant du processus appelant. Bien que définie dans SUSv3, cette fonction n'est pas portable sur tous les systèmes BSD.

exemple_getsid.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv[])
{
    int     i;
    long int pid;
    long int sid;

    if (argc == 1) {
        fprintf(stdout, "%d : %d\n", getpid(), getsid(0));
        return 0;
```

Capacités d'un processus

Depuis Linux 2.2, la toute-puissance d'un processus exécuté sous l'UID effectif root peut être limitée. Une application dispose à présent d'un jeu de capacités permettant de définir ce que le processus peut faire sur le système. Cela est défini dans le document Posix.1e (anciennement Posix.6).

Les capacités d'un processus correspondent à des privilèges, aussi les applications courantes ont-elles des ensembles de capacités vides. En dotant un programme d'un jeu restreint de privilèges (par exemple pour modifier sa propre priorité d'ordonnancement, on lui accorde une puissance suffisante pour accomplir son travail, tout en évitant tout problème de sécurité qui pourrait survenir si le programme était détourné de son utilisation normale. Ainsi, même si une faille de sécurité existe dans l'application, et si elle est découverte par un utilisateur malintentionné, celui-ci ne pourra exploiter que le privilège accordé au programme et pas d'autres capacités dangereuses réservées habituellement à root (par exemple pour insérer un module personnel dans le noyau).

Un processus dispose de trois ensembles de capacités :

- L'ensemble des capacités effectives est celui qui est utilisé à un instant donné pour vérifier les autorisations du processus. Cet ensemble joue un rôle similaire à celui de l'UID effectif, qui n'est pas nécessairement égal à l'UID réel, mais est utilisé pour les permissions d'accès aux fichiers.
- L'ensemble des capacités transmissibles est celui qui sera hérité lors d'un appel-système `exec()`. Notons que l'appel `fork()` ne modifie pas les ensembles de capacités ; le fils a les mêmes privilèges que son père.
- L'ensemble des capacités possibles est une réserve de privilèges. Un processus peut copier une capacité depuis cet ensemble vers n'importe lequel des deux autres. C'est en fait cet ensemble qui représente la véritable limite des possibilités d'une application.

Une application a le droit de réaliser les opérations suivantes sur ses capacités :

- On peut mettre dans l'ensemble effectif ou l'ensemble transmissible n'importe quelle capacité.
- On peut supprimer une capacité de n'importe quel ensemble.

Un fichier exécutable dispose également en théorie des mêmes trois ensembles. Toutefois, les systèmes de fichier actuels ne permettent pas encore le support pour toutes ces données. Aussi un fichier exécutable Set-UID root est-il automatiquement lancé avec ses ensembles de capacités effectives et possibles remplis. Un fichier exécutable normal démarre avec des ensembles effectif et possible égaux à l'ensemble transmissible du processus qui l'a lancé. Dans tous les cas, l'ensemble transmissible n'est pas modifié durant l'appel-système `exec()`.

Les capacités présentes dans le noyau Linux sont définies dans `<linux/capability.h>`. En voici une description, les astérisques signalant les capacités mentionnées dans le document Posix.1e.

Lorsque nous examinerons une fonction privilégiée, nous indiquerons quelle capacité est nécessaire pour s'en acquitter. Par contre, nous n'allons pas détailler le moyen de configurer les permissions d'un processus, car l'interface du noyau est sujette aux changements. Il existe depuis Linux 2.2 deux appels-système, **`capset()`** et **`capget()`**, permettant de configurer les ensembles de permissions d'un processus. Toutefois, ils ne sont ni portables ni même garantis d'exister dans les noyaux futurs.

Nom	Signification
CAP_CHOWN (*)	Possibilité de modifier le propriétaire ou le groupe d'un fichier.
CAP_DAC_OVERRIDE (*)	Accès complet sur tous les fichiers et les répertoires.
CAP_DAC_READ_SEARCH (*)	Accès en lecture ou exécution sur tous les fichiers et répertoires.
CAP_FOWNER (*)	Possibilité d'agir à notre gré sur un fichier ne nous appartenant pas, sauf pour les cas où CAP_FSETID est nécessaire.
CAP_FSETID (*)	Possibilité de modifier les bits Set-UID ou Set-GID d'un fichier ne nous appartenant pas.
CAP_IPC_LOCK	Autorisation de verrouiller des segments de mémoire partagée et de bloquer des pages en mémoire avec mlock().
CAP_IPC_OWNER	Accès aux communications entre processus sans passer par les autorisations d'accès.
CAP_KILL (*)	Possibilité d'envoyer un signal à un processus ne nous appartenant pas.
CAP_LEASE	Autorisation d'établir un « bail » sur un fichier pour être averti par un signal si un autre processus ouvre ce fichier.
CAP_MKNOD	Création d'un fichier spécial représentant un périphérique.
CAP_LINUX_IMMUTABLE	Modification d'attributs spéciaux des fichiers.
CAP_NET_ADMIN	Possibilité d'effectuer de nombreuses tâches administratives concernant le réseau, les interfaces, les tables de routage, etc.
CAP_NET_BIND_SERVICE	Autorisation d'accéder à un port privilégié sur le réseau (numéro de port inférieur à 1 024).
CAP_NET_BROADCAST	Autorisation d'émettre des données en *broadcast* et de s'inscrire à un groupe *multicast*.
CAP_NET_RAW	Possibilité d'utiliser des sockets réseau de type *raw*.
CAP_SETGID (*)	Autorisation de manipuler le bit Set-GID et de s'ajouter des groupes supplémentaires.
CAP_SETPCAP	Possibilité de transférer nos capacités à un autre processus (impossible à utiliser).
CAP_SETUID (*)	Autorisation de manipuler les bits Set-UID et Set-GID d'un fichier nous appartenant.
CAP_SYS_ADMIN	Possibilité de réaliser de nombreuses opérations de configuration concernant le système proprement dit.
CAP_SYS_BOOT	Autorisation d'arrêter et de redémarrer la machine.
CAP_SYS_CHROOT	Possibilité d'utiliser l'appel-système chroot().
CAP_SYS_MODULE	Autorisation d'insérer ou de retirer des modules de code dans le noyau.
CAP_SYS_NICE	Possibilité de modifier sa priorité d'ordonnancement, ou de basculer en fonctionnement temps-réel.
CAP_SYS_PACCT	Mise en service de la comptabilité des processus.
CAP_SYS_PTRACE	Possibilité de suivre l'exécution de n'importe quel processus.
CAP_SYS_RAWIO	Accès aux ports d'entrée-sortie de la machine.
CAP_SYS_RESOURCE	Possibilité de modifier plusieurs limitations concernant les ressources du système.
CAP_SYS_TIME	Mise à l'heure de l'horloge système.
CAP_SYS_TTY_CONFIG	Autorisation de configurer les consoles.

Pour agir sur les privilèges d'une application, il faut employer la bibliothèque `libcap`, qui n'est pas toujours installée dans les distributions courantes. Cette bibliothèque fournit non seulement des fonctions Posix.1e pour modifier les permissions, mais également des utilitaires permettant, par exemple, de lancer une application avec un jeu restreint de privilèges.

> **Attention**
> Ne confondez pas la bibliothèque `libcap` qui gère les capacités des processus et la bibliothèque `libpcap` qui sert à capturer des paquets sur le réseau.

La segmentation des privilèges habituellement réservés à root est une chose importante pour l'avenir de Linux. Cela permet non seulement à un administrateur de déléguer certaines tâches à des utilisateurs de confiance (par exemple en leur fournissant un shell possédant la capacité `CAP_SYS_BOOT` pour pouvoir arrêter l'ordinateur), mais la sécurité du système est aussi augmentée. Une application ayant besoin de quelques privilèges bien ciblés ne disposera pas de la toute-puissance de root. Ainsi, un serveur X11 ayant besoin d'accéder à la mémoire vidéo aura la capacité `CAP_SYS_RAWIO`, mais ne pourra pas aller écrire dans n'importe quel fichier système. De même, un logiciel d'extraction de pistes audio depuis un CD, comme l'application `cdda2wav`, aura le privilège `CAP_SYS_NICE` car il lui faudra passer sur un ordonnancement temps-réel, mais il n'aura pas d'autres autorisations particulières.

Si un pirate découvre une faille de sécurité lui permettant de faire exécuter le code de son choix sous l'UID effectif de l'application – comme nous le verrons dans le chapitre 10 à propos de la fonction `gets() -`, il n'aura toutefois que le privilège du processus initial. Dans les deux exemples indiqués ci-dessus, il pourra perturber l'affichage grâce à l'accès à la mémoire vidéo, ou bloquer le système en faisant boucler un processus de haute priorité temps-réel. Dans un cas comme dans l'autre, cela ne présente aucun intérêt pour lui. Il ne pourra modifier aucun fichier système (pas d'ajout d'utilisateur, par exemple) ni agir sur le réseau pour se dissimuler en préparant l'attaque d'un autre système. Ses possibilités sont largement restreintes.

Malheureusement, il n'existe pas encore – avec Linux 2.6 – d'interface homogène au niveau des systèmes de fichiers pour configurer les capacités des processus. Des évolutions dans ce sens apparaissent, par exemple le système SeLinux (*Security Enhanced Linux*), mais elles sont encore peu répandues.

Conclusion

Dans ce chapitre, nous avons essayé de définir la notion de processus, la manière d'en créer, et les différents identifiants qui peuvent y être attachés. Une application classique n'a pas souvent l'occasion de manipuler ses UID, GID, etc. Cela devient indispensable toutefois si l'accès à des ressources privilégiées qui doivent être offertes à tout utilisateur est nécessaire. L'application doit savoir perdre temporairement ses privilèges, quitte à les récupérer ultérieurement. De même, certains programmes ayant un dialogue important avec leurs descendants seront amenés à gérer des groupes de processus. Bien entendu, tout ceci est également nécessaire lors de la création de processus démons, comme nous le verrons dans la partie consacrée à la programmation réseau.

Une présentation détaillée des permissions associées aux processus se trouve dans [Bach 1989] *Conception du système Unix*. Nous avons également abordé les principes des capacités Posix.1e qui permettent d'améliorer la sécurité d'une application nécessitant des privilèges. Il faut toutefois être conscient que l'implémentation actuelle de ces capacités est loin d'être aussi riche que ce que propose Posix.1e.

3

Accès à l'environnement

Une application peut être exécutée, sous Unix, dans des contextes très différents. Il existe une multitude de types de terminaux, et le répertoire personnel de l'utilisateur peut se trouver à n'importe quel endroit du système de fichiers (par exemple, les utilisateurs spéciaux *news*, *guest* ou *uucp*). De plus, la plupart des applications permettent une configuration de leur interface en fonction des préférences de l'utilisateur.

Il est donc souvent nécessaire d'avoir accès à différents paramètres de l'environnement dans lequel s'exécute un programme. Pour cela, les systèmes Unix offrent une manière assez élégante de transmettre aux applications des informations relatives aussi bien au système en général (type de système d'exploitation, nom de l'hôte…) qu'à l'utilisateur lui-même (emplacement du répertoire personnel, langage utilisé, fichier contenant le courrier en attente), voire aux paramètres n'ayant trait qu'à la session en cours (type de terminal…).

Nous allons voir dans un premier temps les moyens d'accéder aux variables d'environnement, ainsi qu'une liste des variables les plus couramment utilisées. Nous étudierons par la suite l'accès aux arguments en ligne de commande, comprenant aussi bien les options simples, à la manière SUSv3, que les options longues Gnu. Enfin, nous terminerons ce chapitre en observant un exemple complet de paramétrage d'une application en fonction de son environnement d'exécution.

Variables d'environnement

Les variables d'environnement sont définies sous la forme de chaînes de caractères contenant des affectations du type `NOM=VALEUR`. Ces variables sont accessibles aux processus, tant dans les programmes en langage C que dans les scripts shell, par exemple. Lors de la duplication d'un processus avec un `fork()`, le fils hérite d'une copie des variables d'environnement de son père. Un processus peut modifier, créer ou détruire des variables de son propre environnement, et donc de celui des processus fils à venir, mais en aucun cas il ne peut intervenir sur l'environnement de son père.

Un certain nombre de variables sont automatiquement initialisées par le système lors de la connexion de l'utilisateur. D'autres sont mises en place par les fichiers d'initialisation du shell, d'autres enfin peuvent être utilisées temporairement dans des scripts shell avant de lancer une application.

Lorsqu'un programme C démarre, son environnement est automatiquement copié dans un tableau de chaînes de caractères. Ce tableau est disponible dans la variable globale **environ**, à déclarer ainsi en début de programme (elle n'est pas déclarée dans les fichiers d'en-tête courants) :

```
char ** environ.
```

Ce tableau contient des chaînes de caractères terminées par un caractère nul, et se finit lui-même par un pointeur nul. Chaque chaîne a la forme `NOM=VALEUR`, comme nous l'avons précisé. Voici un exemple de balayage de l'environnement.

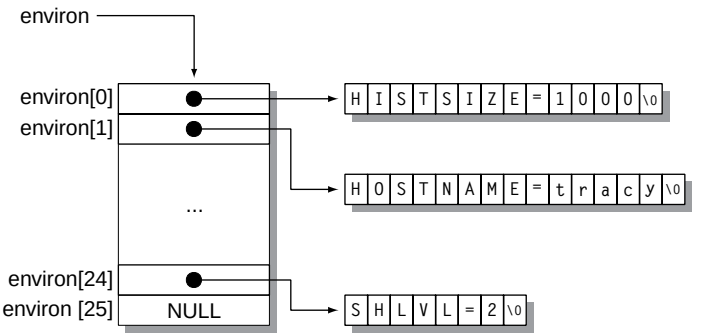
exemple_environ.c :

```
    #include <stdio.h>

    extern char ** environ;

    int
main (void)
{
    int i = 0;

    for (i = 0; environ[i] != NULL; i ++)
        fprintf(stdout, "%d : %s\n", i, environ[i]);

    return 0;
}
```

Voici un exemple d'exécution (raccourci) :

```
$ ./exemple_environ
0 : HISTSIZE=1000
1 : HOSTNAME=tracy
2 : LOGNAME=ccb
3 : HISTFILESIZE=1000
4 : MAIL=/var/spool/mail/ccb
[...]
17 : LC_ALL=fr_FR
18 : DISPLAY=:0.0
19 : LANG=fr_FR
20 : OSTYPE=Linux
21 : MM_CHARSET=ISO-8859-1
22 : WINDOWID=29360142
23 : SHLVL=2
24 : _=./exemple_environ
```

Figure 3.1
Variables d'environnement d'un processus



Notons que le tableau d'environnement est également fourni comme troisième argument à la fonction `main()`, comme les options de ligne de commande `argc` et `argv`, que nous verrons plus bas. La norme SUSv3 recommande, pour des raisons de portabilité, d'éviter d'utiliser cette possibilité et de lui préférer la variable globale `environ`. Voici toutefois un exemple qui fonctionne parfaitement sous Linux.

exemple_environ_2.c :

```
    #include <stdio.h>

    int
main (int argc, char * argv[], char * envp[])
{
    int i = 0;

    for (i = 0; envp[i] != NULL; i ++)
        fprintf(stdout, "%d : %s\n", i, envp[i]);

    return 0;
}
```

On peut parfois avoir besoin de balayer le tableau `environ`, mais c'est assez rare, car les applications ne s'intéressent généralement qu'à un certain nombre de variables bien précises. Pour cela, des fonctions de la bibliothèque C donnent accès aux variables d'environnement afin de pouvoir en ajouter, en détruire, ou en consulter le contenu.

Précisons tout de suite que les chaînes de caractères attendues par les routines de la bibliothèque sont de la forme `NOM=VALEUR`, où il ne doit pas y avoir d'espace avant le signe égal (`=`). En fait, un espace présent à cet endroit serait considéré comme faisant partie du nom de la variable. Notons également que la différenciation entre minuscules et majuscules est prise en compte dans les noms de variables. Les variables d'environnement ont des noms traditionnellement écrits en majuscules (bien que cela ne soit aucunement une obligation), et une chaîne `Home=…` n'est pas considérée comme étant équivalente à `HOME=…`

La routine **`getenv()`** est déclarée dans `<stdlib.h>`, ainsi :

```
char * getenv (const char * nom) ;
```

Elle permet de rechercher une variable d'environnement. On lui donne le nom de la variable désirée, et elle renvoie un pointeur sur la chaîne de caractères suivant immédiatement le signe `=` dans l'affectation `NOM=VALEUR`. Si la variable n'est pas trouvée, la routine renvoie un pointeur NULL.

Avec la GlibC, cette routine renvoie directement un pointeur sur la chaîne de l'environnement du processus. Elle n'effectue pas de copie de la chaîne d'environnement. Aussi, toute modification apportée sur la chaîne renvoyée affectera directement l'environnement du processus comme si on modifiait la variable globale `environ`. D'autres implémentations peuvent préférer recopier le contenu de la chaîne dans une variable globale statique, écrasée à chaque appel.

Précisons bien que la norme SUSv3 interdit toute modification sur le pointeur renvoyé et indique qu'il faut faire une copie de la chaîne renvoyée si on désire la réutiliser par la suite.

Remarquons également que cette fonction n'est ni nécessairement réentrante, ni sûre dans un contexte multi-threads. Nous verrons dans le chapitre 12 les mécanismes de protection (mutex par exemple) permettant de l'utiliser en toute sécurité.

Une variable peut être définie sans avoir de valeur (`NOM=`). Dans ce cas, la routine `getenv()` renverra un pointeur sur une chaîne vide.

exemple_getenv.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (int argc, char * argv[])
{
    int     i;
    char *  variable;

    if (argc == 1) {
        fprintf(stderr, "Utilisation : %s variable...\n", argv[0]);
        return 1;
    }
    for (i = 1; i < argc; i ++) {
        variable = getenv(argv[i]);
        if (variable == NULL)
            fprintf(stdout, "%s : non définie\n", argv[i]);
        else
            fprintf(stdout, "%s : %s\n", argv[i], variable);
    }
    return 0;
}
```

Ce programme permet de tester la valeur des variables d'environnement dont on lui transmet le nom sur la ligne de commande. Nous étudierons plus loin le fonctionnement des arguments `argc` et `argv`.

```
$ ./exemple_getenv HOME LANG SHELL INEXISTANTE
HOME : /home/ccb
LANG : fr_FR
SHELL : /bin/bash
INEXISTANTE : non définie
$
```

Pour tester nos programmes, il est intéressant de voir comment remplir les variables d'environnement au niveau du shell. Cela dépend bien entendu du type d'interpréteur de commandes utilisé. Certains shells font une différence entre leurs propres variables (qu'on utilise pour stocker des informations dans les scripts) et les variables de l'environnement qui seront transmises aux processus fils. Voici les syntaxes pour les principaux interpréteurs de commandes utilisés sous Linux :

Avec les shells bash ou ksh :

Assignation d'une variable du shell :

```
NOM=VALEUR
```

Visualisation d'une variable du shell :

```
echo $NOM
```

Visualisation de toutes les variables définies (internes au shell et environnement) :

```
set
```

Visualisation des variables d'environnement :

```
env
```

Exportation de la variable vers l'environnement :

```
export NOM
```

ou directement :

```
export NOM=VALEUR
```

Destruction d'une variable :

```
unset NOM
```

Avec le shell tcsh :

Assignation d'une variable pour le shell uniquement :

```
set NOM=VALEUR
```

Assignation d'une variable de l'environnement :

```
setenv NOM VALEUR
```

Visualisation de la valeur d'une variable de l'environnement :

```
printenv NOM
```

Destruction d'une variable d'environnement :

```
unsetenv NOM
```

Les exemples que nous donnerons seront réalisés avec `bash`, mais on pourra facilement les transformer pour d'autres shells.

```
$ ESSAI=UN
$ ./exemple_getenv ESSAI
ESSAI : non définie
```

```
$ export ESSAI
$ ./exemple_getenv ESSAI
ESSAI : UN
$ unset ESSAI
$ ./exemple_getenv ESSAI
ESSAI : non définie
$ export ESSAI=DEUX
$ export VIDE=
$ ./exemple_getenv ESSAI VIDE
ESSAI : DEUX
VIDE :
$
```

Les routines **putenv()** et **setenv()** servent à créer une variable d'environnement ou à en modifier le contenu. Elles sont toutes deux déclarées dans `<stdlib.h>` :

```
int putenv (const char * chaine) ;
int setenv (const char * nom, const char * valeur, int ecraser) ;
```

La fonction `putenv()` ne prend qu'un seul argument, une chaîne du type `NOM=VALEUR`, et fait appel à `setenv()` après avoir séparé les deux éléments de l'affectation.

La routine `setenv()` prend trois arguments : les deux premiers sont les chaînes `NOM` et `VALEUR`, et le troisième est un entier indiquant si la variable doit être écrasée dans le cas où elle existe déjà. Le fait d'utiliser un troisième argument nul permet de configurer, en début d'application, des valeurs par défaut, qui ne seront prises en compte que si la variable n'est pas déjà remplie.

Ces deux routines renvoient zéro si elle réussissent, ou –1 s'il n'y a pas assez de mémoire pour créer la nouvelle variable.

La routine **unsetenv()** permet de supprimer une variable :

```
void unsetenv (const char * nom) ;
```

Cette routine recherche la variable dont le nom lui est transmis, l'efface si elle la trouve, et ne renvoie rien.

Un effet de bord – discutable – de la fonction `putenv()`, fournie par la bibliothèque GlibC, est le suivant : si la chaîne transmise à `putenv()` ne contient pas de signe égal (`=`), cette dernière est considérée comme le nom d'une variable, qui est alors supprimée de l'environnement en invoquant `unsetenv()`.

Les routines `getenv()`, `setenv()` et `unsetenv()` de la bibliothèque GlibC balayent le tableau d'environnement pour rechercher la variable désirée en utilisant la fonction `strncmp()`. Elles sont donc sensibles, comme nous l'avons déjà précisé, aux différences entre majuscules et minuscules dans les noms de variables.

Notons l'existence, avec GlibC, d'une routine **clearenv()**, déclarée dans `<stdlib.h>`. Cette routine n'a finalement pas été définie dans la norme SUSv3 et reste donc d'une portabilité limitée. Elle sert à effacer totalement l'environnement du processus appelant (ce qui présente vraiment peu d'intérêt pour une application classique).

Les modifications apportées par un programme C ne jouent que dans son environnement – et celui de ses futurs et éventuels descendants –, mais pas dans celui de son processus père (le shell). Pour visualiser l'action des routines décrites ci-dessus, nous devrons donc écrire un programme un peu plus long que d'habitude.

exemple_putenv.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    void    recherche_variable (char * nom);

    int
main (void)
{
    fprintf(stdout, "\n--- test de putenv() --- \n");
    recherche_variable("ESSAI");
    fprintf(stdout, "putenv(\"ESSAI=UN\");\n");
    putenv("ESSAI=UN");
    recherche_variable("ESSAI");
    fprintf(stdout, "putenv(\"ESSAI=\");\n");
    putenv("ESSAI=");
    recherche_variable("ESSAI");
    fprintf(stdout, "putenv(\"ESSAI\"); équivaut à unsetenv()\n");
    putenv("ESSAI");
    recherche_variable("ESSAI");

    fprintf(stdout, "\n--- test de setenv() --- \n");
    recherche_variable("ESSAI");
    fprintf(stdout, "setenv(\"ESSAI\", \"DEUX\", 1);\n");
    setenv("ESSAI", "DEUX", 1);
    recherche_variable("ESSAI");
    fprintf(stdout, "setenv(\"ESSAI\", \"TROIS\", 1);\n");
    setenv("ESSAI", "TROIS", 1);
    recherche_variable("ESSAI");
    fprintf(stdout, "setenv(\"ESSAI\", \"QUATRE\", 0);"
                "  écrasement de valeur non autorisé\n");
    setenv("ESSAI", "QUATRE", 0);
    recherche_variable("ESSAI");

    fprintf(stdout, "\n-- test de unsetenv() -- \n");
    recherche_variable("ESSAI");
    fprintf(stdout, "unsetenv(\"ESSAI\");\n");
    unsetenv("ESSAI");
    recherche_variable("ESSAI");

    return 0;
}

    void
recherche_variable (char * nom)
{
    char * valeur;
    fprintf(stdout, "   variable %s ", nom);
    valeur = getenv(nom);
    if (valeur == NULL)
        fprintf(stdout, "inexistante\n");
```

```
    else
        fprintf(stdout, "= %s\n", valeur);
}
```

Et voici un exemple d'exécution :

```
$ ./exemple_putenv

--- test de putenv() ---
   variable ESSAI inexistante
putenv("ESSAI=UN");
   variable ESSAI = UN
putenv("ESSAI=");
   variable ESSAI =
putenv("ESSAI"); équivaut à unsetenv()
   variable ESSAI inexistante

--- test de setenv() ---
   variable ESSAI inexistante
setenv("ESSAI", "DEUX", 1);
   variable ESSAI = DEUX
setenv("ESSAI", "TROIS", 1);
   variable ESSAI = TROIS
setenv("ESSAI", "QUATRE", 0); écrasement de valeur non autorisé
   variable ESSAI = TROIS

-- test de unsetenv() --
   variable ESSAI = TROIS
unsetenv("ESSAI");
   variable ESSAI inexistante
$
```

Variables d'environnement couramment utilisées

Un certain nombre de variables sont toujours disponibles sur les machines Linux et peuvent être employées par les applications désirant s'informer sur le système dans lequel elles s'exécutent. Pour voir comment l'environnement des processus est constitué, il est intéressant de suivre leur héritage depuis le démarrage du système.

À tout seigneur tout honneur, le noyau lui-même commence par remplir l'environnement du processus initial (qui deviendra ensuite `init`) avec les chaînes suivantes (dans `/usr/src/linux/init/main.c`) :

```
HOME=/
TERM=linux
```

Le noyau recherche le fichier `init` dans les emplacements successifs suivants : `/sbin/init`, `/etc/init` et `/bin/init`. Puis, il le lance.

Le fichier `/sbin/init` est généralement fourni sous Linux, aussi bien sur les systèmes Red Hat que Slackware, ou Debian, dans le package SysVinit, qui comprend un certain nombre d'utilitaires comme `init`, `shutdown`, `halt`, `last` ou `reboot`.

Ce programme `init` configure plusieurs variables d'environnement.

```
PATH=/usr/local/sbin:/sbin:/bin:/usr/sbin:/usr/bin
RUNLEVEL=niveau d'exécution
PREVLEVEL=niveau précédent (en cas de redémarrage à chaud)
CONSOLE=périphérique console
```

Ensuite, il analyse le fichier `/etc/inittab` et en décode les différents champs. Nous allons suivre simplement l'exemple d'une connexion sur un terminal virtuel, décrite par une ligne :

```
1:12345:respawn:/sbin/mingetty tty1
```

Dans cette configuration, c'est le programme `mingetty` qui est utilisé pour surveiller la ligne de connexion (`tty1`) et déclencher ensuite `/bin/login`. Au passage, `mingetty` configure la variable :

```
TERM=linux
```

Le programme `/bin/login` appartient au package util-linux, qui contient un nombre important d'utilitaires. Ce programme commence par vérifier l'identité de l'utilisateur et en déduit son shell de connexion (la plupart du temps grâce au fichier `/etc/passwd`). Si `login` a reçu l'option `-p` en argument, il conserve l'environnement original, sinon il le détruit en conservant la variable `TERM`. Ensuite, il configure les variables suivantes :

```
HOME=répertoire de l'utilisateur (lu dans /etc/passwd)
SHELL=shell de connexion (idem)
TERM=linux (inchangé)
PATH=/usr/bin:/bin (déclaré par la constante _PATH_DEFPATH dans <paths.h>)
MAIL=emplacement du fichier de boîte à lettres de l'utilisateur
LOGNAME=nom de l'utilisateur
USER=nom de l'utilisateur
```

La redondance des deux dernières variables s'explique par la différence de comportement entre les programmes de type BSD (qui préfèrent `USER`) et ceux de type Système V, qui utilisent `LOGNAME`.

Le programme `/bin/login` lance ensuite le shell choisi par l'utilisateur dans le fichier `/etc/passwd`. Le shell configure lui-même un certain nombre de variables d'environnement dépendant de l'interpréteur. Enfin, il lit certains fichiers d'initialisation pouvant eux-mêmes contenir des affectations de variables d'environnement. Ces fichiers peuvent être généraux pour le système (par exemple, `/etc/profile`) ou spécifiques à l'utilisateur (`~/.profile`). Leurs noms peuvent également varier en fonction du shell utilisé.

En plus des variables d'environnement « classiques » que nous allons voir ci-dessous, une application peut très bien faire varier son comportement en fonction de variables qui lui sont tout à fait propres. Une application foo peut rechercher ses fichiers de configuration dans le répertoire signalé dans la variable `FOODIR`, et créer ses fichiers temporaires dans le répertoire indiqué dans la variable `FOOTMP`. Bien entendu, si ces variables n'existent pas, l'application devra prévoir des valeurs par défaut. Il sera alors plus facile pour l'utilisateur de se créer un script shell de lancement de l'application (par exemple avec bash) :

```
#! /bin/sh

export FOODIR=/usr/local/lib/foo/
export FOOCFG=$HOME/.foo/
```

```
export FOOTMP=/tmp/foo/

exec /usr/local/bin/foo
```

Les variables d'environnement les plus couramment utilisées sont les suivantes :

- `HOME` contient le répertoire personnel de l'utilisateur.
- `PATH` indique la liste des répertoires où on recherche les fichiers exécutables. Ces répertoires sont séparés par des deux-points ':'.
- `PWD` correspond au répertoire de travail du shell lors du lancement de l'application.
- `LANG` indique la localisation choisie par l'utilisateur, complétée par les variables `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`. Ces variables seront détaillées dans le chapitre consacré à l'internationalisation.
- `LOGNAME` et/ou `USER` contiennent le nom de l'utilisateur.
- `TERM` correspond au type de terminal utilisé.
- `SHELL` indique le shell de connexion de l'utilisateur.

D'autres variables sont plutôt liées au comportement de certaines routines de bibliothèque, comme :

- `TMPDIR` est analysée par les routines `tempnam()`, `tmpnam()`, `tmpfile()`, etc.
- `POSIXLY_CORRECT` modifie le comportement de certaines routines pour qu'elles soient strictement conformes à la norme Posix (SUSv3). Ainsi `getopt()`, que nous verrons plus bas, agit différemment suivant que la variable est définie ou non avec les arguments qu'elle rencontre sur la ligne de commande et qui ne représentent pas des options valides.
- `MALLOC_xxx` représente toute une famille de fonctions permettant de contrôler le comportement des routines d'allocation mémoire du type `malloc()`.
- `TZ` correspond au fuseau horaire et modifie le comportement de `tzset()`.

Bien entendu, le comportement de nombreuses routines est influencé par les variables de localisation `LC_xxx`.

Lorsqu'une application utilise les variables d'environnement pour adapter son comportement, il est très fortement recommandé de bien documenter l'utilisation qu'elle en fait (dans la section Environnement de sa page de manuel, par exemple).

Arguments en ligne de commande

Les programmes en langage C reçoivent traditionnellement, dans un tableau de chaînes de caractères, les arguments qui leur sont transmis sur leur ligne de commande. Le nombre d'éléments de ce tableau est passé en premier argument de la fonction `main()`, et le tableau est transmis en second argument. Ces deux éléments sont habituellement notés **`argc`** (args count, nombre d'arguments) et **`argv`** (args values, valeurs des arguments).

Normalement, un programme reçoit en première position du tableau `argv` (donc à la position 0) son propre nom de fichier exécutable.

Lorsqu'une application est lancée par un shell, la ligne de commande est analysée et découpée en arguments en utilisant comme séparateurs certains caractères spéciaux. Par exemple, avec `bash` la liste de ces caractères est conservée dans la variable d'environnement `IFS` et contient l'espace, la tabulation et le retour chariot.

Une application peut donc parcourir sa ligne de commande.

exemple_argv.c :

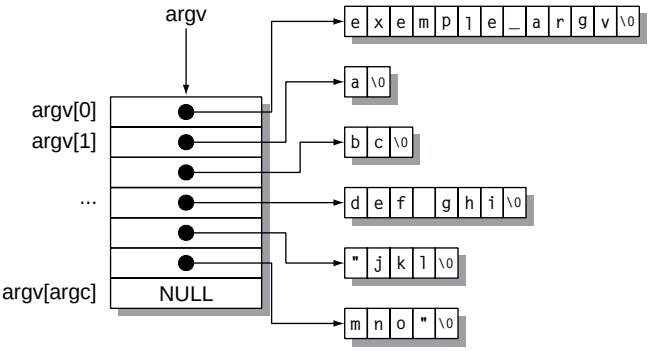
```
    #include <stdio.h>

    int
main (int argc, char * argv [])
{
    int i;

    fprintf(stdout, "%s a reçu en argument :\n", argv[0]);
    for (i = 1; i < argc; i ++)
        fprintf(stdout, "  %s\n", argv[i]);
    return 0;
}
```

Voici un exemple d'exécution, montrant que le shell a considéré comme un argument unique l'ensemble `"def ghi"`, y compris l'espace, grâce à la protection qu'offraient les guillemets, mais que celle-ci est supprimée lorsqu'on fait précéder les caractères d'un antislash (\) et qu'ils deviennent alors des caractères normaux :

```
$ ./exemple_argv a bc "def ghi" \"jkl mno\"
./exemple_argv a reçu en argument :
  a
  bc
  def ghi
  "jkl
  mno"
$
```

Par convention, le tableau `argv []` contient `(argc + 1)` éléments, le dernier étant un pointeur `NULL`.

Figure 3.2
Arguments en ligne de commande du processus



Certains programmes peuvent parfaitement se contenter d'analyser ainsi leur ligne de commande, surtout si on ne doit y trouver qu'un nombre fixe d'arguments (par exemple, uniquement un nom de fichier à traiter), et si aucune option n'est prévue pour modifier le déroulement du processus.

Toutefois, la plupart des applications permettent à l'utilisateur d'indiquer des options en ligne de commande et de fournir de surcroît des arguments qui ne sont pas des options. Nous faisons ici la distinction entre les options du type `-v -r -f`, etc., qu'on trouve dans la plupart des utilitaires Unix, et les autres arguments, comme les noms de fichiers à copier pour la commande `cp`. La bibliothèque GlibC offre des fonctions puissantes pour l'analyse automatique de la ligne de commande afin d'en extraire les arguments qui représentent des options.

Signalons aussi que certaines options prennent elles-mêmes un argument. Par exemple, l'option `-S` de la version GNU de `cp` réclame un argument représentant le suffixe à utiliser pour conserver une copie de secours des fichiers écrasés.

Options simples – SUSv3

Les options, à la manière SUSv3, sont précédées d'un tiret (`-`), et sont représentées par un caractère alphanumérique simple. On peut toutefois regrouper plusieurs options à la suite du même tiret (par exemple, `-a -b -c` équivalent à `-abc`). Si une option nécessite un argument, elle peut en être séparée ou non par un espace (`-a fichier` équivaut à `-afichier`).

L'option spéciale « `--` » (deux tirets) sert à indiquer la fin de la liste des options. Tous les arguments à la suite ne seront pas considérés comme des options. On peut ainsi se débarrasser d'un fichier nommé « `-f` » avec la commande `rm -- -f`.

Un tiret isolé n'est pas considéré comme une option. Il est transmis au programme comme un argument non option.

Normalement, l'utilisateur doit fournir d'abord les options sur sa ligne de commande, et ensuite uniquement les autres arguments. Toutefois, la bibliothèque GlibC réordonne au besoin les arguments de la ligne de commande.

Pour lire aisément les options fournies à une application, la bibliothèque C offre la fonction **`getopt()`** et les variables globales **`optind`**, **`opterr`**, **`optopt`** et **`optarg`**, déclarées dans `<unistd.h>` :

```
int getopt (int argc, const char * argv [], const char * options);
extern int optind ;
extern int opterr ;
extern int optopt ;
extern char * optarg ;
```

On transmet à la fonction `getopt()` les arguments `argc` et `argv` qu'on a reçus dans la fonction `main()`, ainsi qu'une chaîne de caractères indiquant les options reconnues par le programme. À chaque invocation de la fonction, celle-ci nous renverra le caractère correspondant à l'option en cours, et la variable globale externe `optarg` pointera vers l'éventuel argument de la fonction. Lorsque toutes les options auront été parcourues, `getopt()` nous renverra –1, et la variable externe `optind` contiendra le rang du premier élément de `argv[]` qui ne soit pas une option.

Si `getopt()` rencontre un caractère d'option non reconnu, elle affiche un message sur le flux `stderr`. Si la variable externe globale `opterr` ne contient pas 0, elle copie le caractère inconnu dans la variable globale externe `optopt` et renvoie le caractère '`?`'.

La chaîne de caractères qu'on transmet en troisième argument à `getopt()` contient la liste de tous les caractères d'option reconnus. Si une option prend un argument, on fait suivre le caractère d'un deux-points '`:`'.

Voici un premier exemple d'analyse des options en ligne de commande, dans lequel le programme reconnaît les options `a`, `b`, `X`, `Y` seules, et l'option `-c` suivie d'un argument. Si un caractère d'option n'est pas reconnu, nous gérerons nous-même l'affichage d'un message d'erreur. Enfin, une fois terminée l'analyse des options, nous afficherons un à un les arguments restants (qui pourraient représenter par exemple des noms de fichiers à traiter).

exemple_getopt.c :

```
    #include <stdio.h>
    #include <unistd.h>

    int
main (int argc, char * argv [])
{

    char * liste_options = "abc:XY";
    int    option;

    opterr = 0; /* Pas de message d'erreur automatique */

    while ((option = getopt(argc, argv, liste_options)) != -1) {
        switch (option) {
            case 'a' :
                fprintf(stdout, "Option a\n");
                break;
            case 'b' :
                fprintf(stdout, "Option b\n");
                break;
            case 'c' :
                fprintf(stdout, "Option c %s\n", optarg);
                break;
            case 'X' :

            case 'Y' :
                fprintf(stdout, "Option %c\n", option);
                break;
            case '?' :
                fprintf(stderr, "Option %c fausse\n", optopt);
                break;
        }
    }

    if (optind != argc) {
        fprintf(stdout, "Arguments restants :\n");

        while (optind != argc)
            fprintf(stdout, "  %s\n", argv[optind ++]);
    }

    return 0;
}
```

Voici un exemple d'exécution regroupant une bonne partie des fonctionnalités disponibles avec `getopt()` :

```
$ ./exemple_getopt -abd -c 12 -XY suite et fin
Option a
Option b
Option d fausse
Option c 12
Option X
Option Y
Arguments restants :
  suite
  et
  fin
$
```

La variable externe globale `optarg`, qu'on utilise pour accéder à l'argument de certaines options, est en réalité un pointeur, de type `char *`, dirigé vers l'élément de `argv[]` qui correspond à la valeur désirée. Il n'est donc pas nécessaire de copier la chaîne de caractères si on désire l'utiliser plus tard ; on peut directement copier la valeur du pointeur, puisque le tableau `argv[]` ne doit plus varier après l'invocation de `getopt()`. Nous verrons un exemple plus concret d'utilisation de cette chaîne de caractères dans le programme nommé `exemple_options.c`, fourni à la fin de ce chapitre.

Options longues – Gnu

Les applications issues du projet Gnu ont ajouté un autre type d'options qui ont été incorporées dans les routines d'analyse de la ligne de commande : les options longues. Il s'agit d'options commençant par deux tirets « `--` », et dont le libellé est exprimé par des mots complets. Par exemple, la version Gnu de `ls` accepte l'option longue `--numeric-uid-gid` de manière équivalente à `-n`.

Bien entendu, ces options ne sont pas prévues pour être utilisées quotidiennement en ligne de commande. Peu d'utilisateurs préfèrent saisir

```
ln --symbolic --force foo bar
```

à la place de

```
ln -sf foo bar
```

Par contre, ces options longues sont très commodes lorsqu'elles sont utilisées dans un script shell, où elles permettent d'auto-documenter les arguments fournis à une commande peu utilisée.

Les options longues peuvent bien entendu accepter des arguments, qui s'écrivent aussi bien

```
--option valeur
```

que

```
--option=valeur
```

Une option longue peut être abrégée tant qu'il n'y a pas d'ambiguïté avec d'autres options de la même commande. La bibliothèque GlibC offre des routines d'analyse des options longues

assez semblables à la routine `getopt()` ; il s'agit de `getopt_long()` et de `getopt_long_only()`. Ces routines sont déclarées dans le fichier d'en-tête `<getopt.h>` et non dans `<unistd.h>`. La fonction **`getopt_long()`** a le prototype suivant :

```
int getopt_long (int argc, char * argv [],
                 const char * optsring,
                 const struct option * longopts,
                 int * longindex);
```

Attention toutefois aux problèmes de portabilité : même si elle n'existe pas sur tous les systèmes, la routine `getopt()` est définie par SUSv3 et est donc très répandue sous Unix. Par contre, les options longues (et même le fichier d'en-tête `<getopt.h>`) sont des extensions Gnu largement moins courantes. Si une application doit être portable sous plusieurs systèmes Unix, il est conseillé d'encadrer les portions de code spécifiques aux options longues par des directives `#ifdef` / `#endif` permettant à la compilation de basculer au choix avec ou sans options longues.

La routine `getopt_long()` prend `argc` et `argv[]` en premiers arguments comme `getopt()`. Ensuite, on lui transmet également une chaîne de caractères contenant les options courtes, exactement comme `getopt()`. Puis viennent deux arguments supplémentaires : un tableau d'objets de type `struct option`, et un pointeur sur un entier. La structure `struct option` est définie dans le fichier d'en-tête `<getopt.h>` ainsi :

Nom	Type	Signification
`name`	`char *`	Nom de l'option longue.
`has_arg`	`int`	L'option réclame-t-elle un argument supplémentaire ?
`flag`	`int`	Manière de renvoyer la valeur ci-dessous.
`val`	`int`	Valeur à renvoyer quand l'option est trouvée.

Chaque élément du tableau `longopts` contient une option longue, le dernier élément devant être obligatoirement rempli avec des zéros.

Le premier champ comprend simplement le nom de l'option. C'est une chaîne de caractères classique, terminée par un caractère nul. Le second champ indique si l'option doit être suivie par un argument. Il y a trois possibilités, décrites par des constantes symboliques dans le fichier `<getopt.h>` :

- `no_argument` (0) : l'option ne prend pas d'argument.
- `required_argument` (1) : l'option prend toujours un argument.
- `optional_argument` (2) : l'argument est éventuel.

Le troisième champ est plus compliqué. S'il est `NULL` (c'est le cas le plus courant), l'appel à `getopt_long()` renverra, lorsqu'il trouvera l'option, la valeur indiquée dans le champ `val`. Ce principe est donc assez semblable à celui qu'on a déjà vu pour `getopt()`, et il est même habituel de mettre dans le champ `val` le caractère correspondant à l'option courte équivalente, afin d'avoir un traitement `switch/case` unique. Dans le cas où ce troisième champ (`flag`) n'est pas `NULL`, il faut le faire pointer vers une variable de type `int`, par exemple une variable déclarée dans la fonction `main()`, dans laquelle `getopt_long()` écrira la valeur contenue dans le champ `val` si l'option est rencontrée. Dans un tel cas, `getopt_long()` renvoie 0.

Lorsque `getopt_long()` rencontre une option courte (contenue dans la chaîne `optstring`), elle se comporte exactement comme `getopt()`. Lorsqu'elle rencontre une option longue, elle remplit la variable pointée par `longindex` avec l'indice de l'option en question dans le tableau `longopts`. Comme pour les options courtes, les arguments éventuels sont transmis par le pointeur global `optarg`. Celui-ci est `NULL` si l'option n'a pas d'argument (ce qui sert dans le cas d'arguments optionnels).

Pour remplir le tableau `longopts` que nous devons fournir à `getopt_long()`, il est pratique d'utiliser l'initialisation automatique d'une variable statique de la fonction `main()`. Nous allons écrire un petit programme (qu'on peut imaginer comme un lecteur de fichiers vidéo) acceptant les options suivantes :

- `--debut` ou `-d`, suivie d'une valeur numérique entière
- `--fin` ou `-f`, suivie d'une valeur numérique entière
- `--rapide`
- `--lent`

Les deux dernières options serviront à mettre directement à jour une variable interne du programme, en utilisant un champ `flag` non `NULL`. Nous ne traitons pas dans ce programme les arguments autres que les options (une fois que `getopt_long()` renvoie –1), et nous laissons à cette routine le soin d'afficher un message d'erreur en cas d'option non reconnue.

exemple_getopt_long.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <getopt.h>

    int     vitesse_lecture = 0;
  /* -1 = lent, 0 = normal, 1 = rapide */

    int
main (int argc, char * argv[])
{
    char * optstring = "d:f:";
    struct option longopts[] = {

    /* name         has_arg flag                    val */
    { "debut",      1,      NULL,                   'd' },
    { "fin",        1,      NULL,                   'f' },
    { "rapide",     0,      & vitesse_lecture,       1  },
    { "lent",       0,      & vitesse_lecture,      -1  },
    /* Le dernier élément doit être nul */
    { NULL,         0,      NULL,                    0},
    };

    int     longindex;
    int     option;
    int     debut = 0;
    int     fin = 999;
```

```
    while ((option = getopt_long(argc, argv,
                    optstring, longopts, & longindex)) != -1) {

        switch (option) {

            case 'd' :
                if (sscanf(optarg, "%d", & debut) != 1) {
                    fprintf(stderr, "Erreur pour début\n");
                }
                break;
            case 'f' :
                if (sscanf(optarg, "%d", & fin) != 1) {
                    fprintf(stderr, "Erreur pour fin\n");
                }
                break;
            case 0 :
                /* vitesse_lecture traitée automatiquement */
                break;
            case '?' :
                /*  On a laissé opterr à 1 */
                break;
        }
    }
    fprintf(stdout, "Vitesse %d, début %d, fin %d\n",
                        vitesse_lecture, debut, fin);
    return 0;
}
```

En voici un exemple d'exécution :

```
$ ./exemple_getopt_long  --rapide -d 4 --fin 25
Vitesse 1, début 4, fin 25
$
```

Il existe également avec la GlibC une routine **getopt_long_only()** fonctionnant comme `getopt_long()`, à la différence que même une option commençant par un seul tiret (`-`) est considérée d'abord comme une option longue puis, en cas d'échec, comme une option courte. Cela signifie que `-ab` sera d'abord considérée comme équivalant à `--ab` (donc comme une abréviation de `--abort`) avant d'être traitée comme la succession d'options simples `-a -b`. Cet usage peut induire l'utilisateur en erreur, et cette routine me semble peu recommandable…

Sous-options

L'argument qu'on fournit à une option peut parfois nécessiter lui-même une analyse pour être séparé en sous-options. La bibliothèque C fournit dans `<stdlib.h>` une fonction ayant ce rôle : **getsubopt()**. La déclaration n'est présente dans le fichier d'en-tête que si la constante symbolique `_XOPEN_SOURCE` est définie et contient la valeur 500, ou si la constante `_GNU_SOURCE` est définie.

L'exemple classique d'utilisation de cette fonction est l'option `-o` de la commande `mount`. Cette option est suivie de n'importe quelle liste de sous-options séparées par des virgules, certaines pouvant prendre une valeur (par exemple `-o async,noexec,bs=512`).

Le prototype de `getsubopt()` est le suivant :

```
int getsubopt (char ** option, const char * const * tokens,
               char ** value);
```

Cette routine n'est appelée que lorsqu'on se trouve dans le `case` correspondant à l'option à analyser de nouveau (par `-o` pour `mount`). Il faut transmettre un pointeur en premier argument sur un pointeur contenant la sous-option. En d'autres termes, on crée un pointeur `char * subopt` qu'on fait pointer sur la chaîne à analyser (`subopt = optarg`), et on transmet `& subopt` à la fonction. Celle-ci avancera ce pointeur d'une sous-option à chaque appel, jusqu'à ce qu'il arrive sur le caractère nul de fin de `optarg`.

Le second argument est un tableau contenant des chaînes de caractères correspondant aux sous-options. Le dernier élément de ce tableau doit être un pointeur `NULL`.

Enfin, on transmet en dernier argument l'adresse d'un pointeur de chaîne de caractères. Lorsque la routine rencontre une sous-option suivie d'un signe égal '=', elle renseigne ce pointeur de manière à l'amener au début de la valeur. Elle inscrit également un caractère nul pour marquer la fin de la valeur. Si aucune valeur n'est disponible, `value` est rempli avec `NULL`.

Si une sous-option est reconnue, son index dans la table `tokens` est renvoyé. Sinon, `getsubopt()` renvoie –1. Un exemple de code permettant l'analyse d'une sous-option sera fourni dans le programme `exemple_options.c` décrit ci-après.

Exemple complet d'accès à l'environnement

Nous allons voir un exemple de code permettant de regrouper l'ensemble des fonctionnalités d'accès à l'environnement que nous avons vues dans ce chapitre. Nous allons imaginer qu'il s'agit d'une application se connectant par exemple sur un serveur TCP/IP, comme nous aurons l'occasion d'en étudier plus loin.

Notre application doit fournir tout d'abord des valeurs par défaut pour tous les éléments paramétrables. Ces valeurs sont établies à la compilation du programme. Toutefois, on les regroupe toutes ensemble afin que l'administrateur du système puisse, s'il le désire, recompiler l'application avec de nouvelles valeurs par défaut.

Ensuite, nous essaierons d'obtenir des informations en provenance des variables d'environnement. Celles-ci peuvent être renseignées par l'administrateur système (par exemple dans `/etc/profile`) ou par l'utilisateur (dans `~/.profile` ou dans un script shell de lancement de l'application).

Puis, nous analyserons la ligne de commande. Il est en effet important que les options fournies manuellement par l'utilisateur aient la priorité sur celles qui ont été choisies pour l'ensemble du système.

Voyons la liste des éléments dont nous allons permettre le paramétrage.

- Adresse réseau du serveur à contacter

  Il s'agit ici d'une adresse IP numérique ou d'un nom d'hôte. Nous nous contenterons d'obtenir cette adresse dans une chaîne de caractères et de laisser à la suite de l'application les tâches de conversion nécessaires. Nous ne ferons aucune gestion d'erreur sur cette chaîne, nous arrangeant simplement pour qu'elle ne soit pas vide.

Par défaut, la valeur sera `localhost`. On pourra modifier l'adresse en utilisant la variable d'environnement `OPT_ADR`. Les options `-a` et `--adresse`, suivies d'une chaîne de caractères, permettront une dernière configuration.

- Port TCP à utiliser pour joindre le serveur

  Le port TCP sur lequel nous désirons contacter le serveur peut être indiqué soit sous forme numérique, soit sous forme symbolique, en utilisant un nom décrit dans le fichier `/etc/services`. Nous considérerons donc qu'il s'agit d'une chaîne de caractères, que le reste de l'application se chargera de convertir en numéro de port effectif.

  Par défaut, nous prendrons une valeur arbitraire de 4 000, mais nous pourrons modifier cette valeur en utilisant la variable d'environnement `OPT_SRV`, ou l'une des options `-p` ou `--port`, suivie d'une chaîne de caractères.

- Options pour la connexion

  Afin de donner un exemple d'utilisation de la fonction `getsubopt()`, nous allons permettre la transmission d'une liste de sous-options séparées par des virgules, en utilisant l'option `-o` ou `--option` de la ligne de commande :

  `auto` / `nonauto` : il s'agit par exemple de tentative de reconnexion automatique au serveur en cas d'échec de transmission. Ce paramètre est également configurable en définissant (ou non) la variable d'environnement `OPT_AUTO`. Par défaut, le choix est `nonauto`.

  `delai=<duree>` : il s'agit du temps d'attente en secondes entre deux tentatives de reconnexion au serveur. Cette valeur vaut 4 secondes par défaut, mais peut aussi être modifiée par la variable d'environnement `OPT_DELAI`.

- Affichage de l'aide

  Une option `-h` ou `--help` permettra d'obtenir un rappel de la syntaxe de l'application.

- Arguments autres que les options

  Le programme peut être invoqué avec d'autres arguments à la suite des options, par exemple des noms de fichiers à transférer, l'identité de l'utilisateur sur la machine distante, etc. Ces arguments seront affichés par notre application à la suite des options.

Pour lire les sous-options introduites par l'option `-o`, une routine séparée est utilisée, principalement pour éviter des niveaux d'indentation excessifs et inesthétiques en imbriquant deux boucles `while` et deux `switch-case`.

Enfin, pour augmenter la portabilité de notre exemple, nous allons encadrer tout ce qui concerne les options longues Gnu par des directives `#ifdef` - `#else` - `#endif`. Ainsi, la recompilation sera possible sur pratiquement tous les systèmes Unix, à l'exception peut-être de la routine `getsubopt()`. Pour compiler l'application avec les options longues, sous Linux par exemple, il suffira d'inclure une option `-DOPTIONS_LONGUES` sur la ligne de commande de `gcc` (ou dans un fichier `Makefile`). Sur un système où la bibliothèque C n'offre pas la routine `getopt_long()`, il suffira de ne pas définir cette constante symbolique pour permettre la compilation.

exemple_options.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#ifdef OPTIONS_LONGUES
     #include <getopt.h>
#endif

/* Définition des valeurs par défaut.      */
/* (pourraient être regroupées dans un .h) */
    #define ADRESSE_SERVEUR_DEFAUT  "localhost"
    #define PORT_SERVEUR_DEFAUT     "4000"
    #define CONNEXION_AUTO_DEFAUT   0
    #define DELAI_CONNEXION_DEFAUT  4

    void    sous_options      (char * ssopt, int * cnx_auto, int * delai);
    void    suite_application (char * adresse_serveur,
                               char * port_serveur,
                               int    connexion_auto,
                               int    delai_reconnexion,
                               int    argc,
                               char * argv []);
    void    affiche_aide      (char * nom_programme);

    int
main (int argc, char * argv[])
{
    /*
     * Copie des chaînes d'environnement.
     * Il n'est pas indispensable sous Linux d'en faire une
     * copie, mais c'est une bonne habitude pour assurer la
     * portabilité du programme.
     */
    char * opt_adr   = NULL;
    char * opt_srv   = NULL;
    int    opt_delai = 0;
    char * retour_getenv;
    /*
     * Variables contenant les valeurs effectives de nos paramètres.
     */
    static char *   adresse_serveur = ADRESSE_SERVEUR_DEFAUT;
    static char *   port_serveur    = PORT_SERVEUR_DEFAUT;
    int             connexion_auto  = CONNEXION_AUTO_DEFAUT;
    int             delai_connexion = DELAI_CONNEXION_DEFAUT;

    int     option;
    /*
     * Lecture des variables d'environnement, on code en dur ici
     * le nom des variables, mais on pourrait aussi les regrouper
     * (par #define) en tête de fichier.
     */
    retour_getenv = getenv("OPT_ADR");
    if ((retour_getenv != NULL) && (strlen(retour_getenv) != 0)) {
        opt_adr = malloc (strlen(retour_getenv) + 1);
        if (opt_adr != NULL) {
            strcpy(opt_adr, retour_getenv);
```

```
            adresse_serveur = opt_adr;
        } else {
            perror("malloc");
            exit(1);
        }
    }
    retour_getenv = getenv("OPT_SRV");
    if ((retour_getenv != NULL) && (strlen(retour_getenv) != 0)) {
        opt_srv = malloc (strlen(retour_getenv) + 1);
        if (opt_srv != NULL) {
            strcpy(opt_srv, retour_getenv);
            port_serveur = opt_srv;
        } else {
            perror("malloc");
            exit(1);
        }
    }
    retour_getenv = getenv("OPT_AUTO");
    /* Il suffit que la variable existe dans l'environnement,     */
    /* sa valeur ne nous importe pas.                             */
    if (retour_getenv != NULL)
        connexion_auto = 1;
    retour_getenv = getenv("OPT_DELAI");
    if (retour_getenv != NULL)
        if (sscanf(retour_getenv, "%d", & opt_delai) == 1)
            delai_connexion = opt_delai;
    /*
     * On va passer maintenant à la lecture des options en ligne
     * de commande.
     */
    opterr = 1;
    while (1) {
        #ifdef OPTIONS_LONGUES
        int     index = 0;
        static struct option longopts[] = {
            { "adresse",    1,      NULL,   'a' },
            { "port",       1,      NULL,   'p' },
            { "option",     1,      NULL,   'o' },
            { "help",       0,      NULL,   'h' },
            { NULL,         0,      NULL,   0 }
        };
        option = getopt_long(argc, argv, "a:p:o:h", longopts, & index);
        #else
        option = getopt(argc, argv, "a:p:o:h");
        #endif
        if (option == -1)
                break;

        switch (option) {
            case 'a' :
                /* On libère une éventuelle copie de chaîne */
                /* d'environnement équivalente.             */
```

```
                if (opt_adr != NULL)
                    free(opt_adr);
                opt_adr = NULL;
                adresse_serveur = optarg;
                break;
            case ‘p’ :
                /* idem */
                if (opt_srv != NULL)
                    free(opt_srv);
                opt_srv = NULL;
                port_serveur = optarg;
                break;
            case ‘o’ :
                /* on va analyser les sous-options */
                sous_options(optarg,
                            & connexion_auto,
                            & delai_connexion);
                break;
            case ‘h’ :
                affiche_aide(argv[0]);
                exit(0);
            default :
                break;
        }
    }
    suite_application(adresse_serveur, port_serveur, connexion_auto,
                      delai_connexion, argc - optind, & (argv[optind]));
    return 0;
}

    void
sous_options (char * ssopt, int * cnx_auto, int * delai)
{
    int     subopt;
    char *  chaine = ssopt;
    char *  value = NULL;
    int     val_delai;
    char * tokens[] = {
            "auto", "nonauto", "delai", NULL
    };
    while ((subopt = getsubopt(& chaine, tokens, & value)) != -1) {
        switch (subopt) {
            case 0 : /* auto */
                * cnx_auto = 1;
                break;
            case 1 : /* nonauto */
                * cnx_auto = 0;
                break;
            case 2 : /* delai=... */
                if (value == NULL) {
                    fprintf(stderr, "délai attendu\n");
                    break;
```

```
                }
                if (sscanf (value, "%d", & val_delai) != 1) {
                    fprintf(stderr, "délai invalide\n");
                    break;
                }
                * delai = val_delai;
                break;
        }
    }
}
    /*
     * La suite de l'application ne fait qu'afficher
     * les options et les arguments supplémentaires
     */
    void
suite_application (char * adr_serveur,
               char * port_serveur,
               int    cnx_auto,
               int    delai_cnx,
               int    argc,
               char * argv[])
{
    int     i;
    fprintf(stdout, "Serveur : %s - %s\n", adr_serveur, port_serveur);
    fprintf(stdout, "Connexion auto : %s\n", cnx_auto ? "oui":"non");
    fprintf(stdout, "Délai : %d\n", delai_cnx);
    fprintf(stdout, "Arguments supplémentaires : ");
    for (i = 0; i < argc; i++)
        fprintf(stdout, "%s - ", argv[i]);
    fprintf(stdout, "\n");
}

    void
affiche_aide (char * nom_prog)
{
    fprintf(stderr, "Syntaxe : %s [options] [fichiers...]\n", nom_prog);
    fprintf(stderr, "Options :\n");
#ifdef OPTIONS_LONGUES
    fprintf(stderr, "  --help\n");
#endif
    fprintf(stderr, "  -h               Cet écran d'aide \n");
#ifdef OPTIONS_LONGUES
    fprintf(stderr, "  --adresse <serveur> \n");
#endif
    fprintf(stderr, "  -a <serveur>     Adresse IP du serveur \n");
#ifdef OPTIONS_LONGUES
    fprintf(stderr, "  --port <numero_port> \n");
#endif
    fprintf(stderr, "  -p <num_port>    Numéro de port TCP \n");
#ifdef OPTIONS_LONGUES
    fprintf(stderr, "  --option [sous_options]\n");
#endif
```

```
    fprintf(stderr, "  -o [sous_options]     \n");
    fprintf(stderr, "Sous-options :\n");
    fprintf(stderr, "  auto / nonauto   Connexion automatique \n");
    fprintf(stderr, "  delai=<sec>      Délai entre deux connexions \n");
}
```

Voici plusieurs exemples d'utilisation, ainsi que la ligne de commande à utiliser pour définir les constantes nécessaires lors de la compilation :

```
$ cc -D_GNU_SOURCE -DOPTIONS_LONGUES exemple_options.c -o exemple_options
$ ./exemple_options
Serveur : localhost - 4000
Connexion auto : non
Délai : 4
Arguments supplémentaires :
$ export OPT_ADR="172.16.15.1"
$ ./exemple_options
Serveur : 172.16.15.1 - 4000
Connexion auto : non
Délai : 4
Arguments supplémentaires :
$ export OPT_SRV="5000"
$ ./exemple_options --adresse "127.0.0.1"
Serveur : 127.0.0.1 - 5000
Connexion auto : non
Délai : 4
Arguments supplémentaires :
$ export OPT_AUTO=
$ ./exemple_options -p 6000 -odelai=5
Serveur : 172.16.15.1 - 6000
Connexion auto : oui
Délai : 5
Arguments supplémentaires :
$ ./exemple_options -p 6000 -odelai=5,nonauto et d autres arguments
Serveur : 172.16.15.1 - 6000
Connexion auto : non
Délai : 5
Arguments supplémentaires : et - d - autres - arguments
$
```

Conclusion

Nous voici donc en possession d'un squelette complet de programme capable d'accéder à son environnement et permettant un paramétrage à plusieurs niveaux :

- à la compilation, par l'administrateur système, grâce aux valeurs par défaut ;
- globalement pour toutes les exécutions, par l'administrateur ou l'utilisateur, grâce aux variables d'environnement ;
- lors d'une exécution particulière grâce aux options en ligne de commande.

Il est important, pour une application un tant soit peu complète, de permettre ainsi à l'utilisateur et à l'administrateur système de configurer son comportement à divers niveaux.

4

Exécution des programmes

Ce chapitre va être principalement consacré aux débuts d'un processus. Tout d'abord, nous examinerons les méthodes utilisables pour lancer un nouveau programme, ainsi que les mécanismes sous-jacents, qui peuvent conduire à un échec du démarrage.

Nous nous intéresserons ensuite à des fonctions simplifiées, permettant d'utiliser une application indépendante comme une sous-routine de notre logiciel.

Lancement d'un nouveau programme

Nous avons déjà vu que le seul moyen de créer un nouveau processus dans le système est d'invoquer `fork()`, qui duplique le processus appelant. De même, la seule façon d'exécuter un nouveau programme est d'appeler l'une des fonctions de la famille **`exec()`**. Nous verrons également qu'il existe les fonctions `popen()` et `system()`, qui permettent d'exécuter une autre application mais en s'appuyant sur `fork()` et `exec()`.

L'appel de l'une des fonctions `exec()` permet de remplacer l'espace mémoire du processus appelant par le code et les données de la nouvelle application. Ces fonctions ne reviennent qu'en cas d'erreur, sinon le processus appelant est entièrement remplacé.

On parle couramment de l'appel-système `exec()` sous forme générique, mais en fait il n'existe aucune routine ayant ce nom. Simplement, il y a six variantes nommées `execl()`, `execle()`, `execlp()`, `execv()`, `execve()` et `execvp()`. Ces fonctions permettent de lancer une application. Les différences portent sur la manière de transmettre les arguments et l'environnement, et sur la méthode pour accéder au programme à lancer. Il n'existe sous Linux qu'un seul véritable appel-système dans cette famille de fonctions : `execve()`. Les autres fonctions sont implémentées dans la bibliothèque C à partir de cet appel-système.

Les fonctions dont le suffixe commencent par un "`l`" utilisent une liste d'arguments à transmettre de nombre variable, tandis que celles qui débutent par un "`v`" emploient un tableau à la manière du vecteur `argv []`.

Les fonctions se terminant par un "e" transmettent l'environnement dans un tableau envp [] explicitement passé dans les arguments de la fonction, alors que les autres utilisent la variable globale environ.

Les fonctions se finissant par un "p" utilisent la variable d'environnement PATH pour rechercher le répertoire dans lequel se situe l'application à lancer, alors que les autres nécessitent un chemin d'accès complet. La variable PATH est déclarée dans l'environnement comme étant une liste de répertoires séparés par des deux-points. On utilise typiquement une affectation du genre :

```
PATH=/usr/bin:/bin:/usr/X11R6/bin/:/usr/local/bin:/usr/sbin:/sbin
```

Il est préférable de placer en tête de PATH les répertoires dans lesquels se trouvent les applications les plus utilisées afin d'accélérer la recherche. Certains ajoutent à leur PATH un répertoire simplement composé d'un point, représentant le répertoire en cours. Cela peut entraîner une faille de sécurité, surtout si ce répertoire « . » n'est pas placé en dernier dans l'ordre de recherche. Il vaut mieux ne pas le mettre dans le PATH et utiliser explicitement une commande :

```
$ ./mon_prog
```

pour lancer une application qui se trouve dans le répertoire courant.

Quand execlp() ou execvp() rencontrent, lors de leur parcours des répertoires du PATH, un fichier exécutable du nom attendu, ils tentent de le charger. S'il ne s'agit pas d'un fichier binaire mais d'un fichier de texte commençant par une ligne du type :

```
#! /bin/interpreteur
```

le programme indiqué (interpreteur) est chargé, et le fichier lui est transmis sur son entrée standard. Il s'agit souvent de /bin/sh, qui permet de lancer des scripts shell, mais on peut trouver d'autres fichiers à interpréter (/bin/awk, /usr/bin/perl, /usr/bin/wish...). Nous verrons une invocation de script shell plus loin.

Si l'appel exec() réussit, il ne revient pas, sinon il renvoie –1, et errno contient un code expliquant les raisons de l'échec. Celles-ci sont détaillées dans la page de manuel execve(2).

Le prototype de **execve()** est le suivant :

```
int execve (const char * appli, const char * argv [],
            const char * envp []);
```

La chaîne "appli" doit contenir le chemin d'accès au programme à lancer à partir du répertoire de travail en cours ou à partir de la racine du système de fichiers s'il commence par un slash "/".

Le tableau argv[] contient des chaînes de caractères correspondant aux arguments qu'on trouve habituellement sur la ligne de commande.

La première chaîne argv[0] doit contenir le nom de l'application à lancer (sans chemin d'accès). Ceci peut parfois être utilisé pour des applications qui modifient leur comportement en fonction du nom sous lequel elles sont invoquées. Par exemple, /bin/gzip sert à compresser des fichiers. Il est également utilisé pour décompresser des fichiers si on lui transmet l'option -d ou si on l'invoque sous le nom gunzip. Pour ce faire, il analyse argv[0]. Dans la plupart des distributions Linux, il existe d'ailleurs un lien physique nommé /bin/gunzip sur le même fichier que /bin/gzip.

Le troisième argument est un tableau de chaînes déclarant les variables d'environnement. On peut éventuellement utiliser la variable externe globale `environ` si on désire transmettre le même environnement au programme à lancer. Dans la majorité des applications, il est toutefois important de mettre en place un environnement cohérent, grâce aux fonctions que nous avons étudiées dans le chapitre 3. Ceci est particulièrement nécessaire dans les applications susceptibles d'être installées Set-UID *root*.

Les tableaux `argv []` et `envp []` doivent se terminer par des pointeurs `NULL`.

Pour montrer l'utilisation de `execve()`, nous allons invoquer le shell, en lui passant la commande `echo $SHLVL`. Le shell nous affichera alors la valeur de cette variable d'environnement. *bash* comme *tcsh* indiquent dans cette variable le nombre d'invocations successives du shell qui sont « empilées ». Voici un exemple sous *bash* :

```
$ echo $SHLVL
1
$ sh
$ echo $SHLVL
2
$ sh
$ echo $SHLVL
3
$ exit
$ echo $SHLVL
2
$ exit
$ echo $SHLVL
1
$
```

Notre programme exécutera donc simplement cette commande en lui transmettant son propre environnement. On notera que la commande `echo $SHLVL` doit être transmise en un seul argument, comme on le ferait sur la ligne de commande :

```
$ sh -c "echo $SHLVL"
2
$
```

(L'option `-c` demande au shell d'exécuter l'argument suivant, puis de se terminer.)

exemple_execve.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    extern char ** environ;

    int
main (void)
{
    char * argv[] = {"sh", "-c", "echo $SHLVL", (char *) NULL };

    fprintf(stdout, "Je lance /bin/sh -c \"echo $SHLVL\" :\n");
```

```
    execve("/bin/sh", argv, environ);

    fprintf(stdout, "Raté : erreur = %d\n", errno);
    return 0;
}
```

Voici un exemple d'exécution sous *bash* :

```
$ echo $SHLVL
1
$ ./exemple_execve
Je lance /bin/sh -c "echo $SHLVL" :
2
$ sh
$ ./exemple_execve
Je lance /bin/sh -c "echo $SHLVL" :
3
$ exit
$ ./exemple_execve
Je lance /bin/sh -c "echo $SHLVL" :
2
$
```

Bien entendu, le programme ayant lancé un nouveau shell pour exécuter la commande, le niveau d'imbrication est incrémenté par rapport à la variable d'environnement, consultée directement avec `echo $SHLVL`.

La fonction **`execv()`** dispose du prototype suivant :

```
int execv (const char * application, const char * argv []);
```

Elle fonctionne comme `execve()`, mais l'environnement est directement transmis par l'intermédiaire de la variable externe `environ`, sans avoir besoin d'être passé explicitement en argument durant l'appel.

La fonction **`execvp()`** utilise un prototype semblable à celui de `execv()`, mais elle se sert de la variable d'environnement `PATH` pour rechercher l'application. Nous allons en voir un exemple, qui exécute simplement la commande `ls`.

exemple_execvp.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    char * argv[] = { "ls", "-l", "-n", (char *) NULL };

    execvp("ls", argv);

    fprintf(stderr, "Erreur %d\n", errno);
    return 1;
}
```

Lorsqu'on exécute cette application, celle-ci recherche `ls` dans les répertoires de la variable d'environnement `PATH`. Ainsi, en modifiant cette variable pour éliminer le répertoire contenant `ls`, `execvp()` échoue.

```
$ echo $PATH
/usr/bin:/bin:/usr/X11R6/bin:/usr/local/bin:/usr/sbin
$ which ls
/bin/ls
$ ./exemple_execvp
total 12
-rwxrwxr-x   1 500      500          4607 Aug  7 14:53 exemple_execve
-rw-rw-r--   1 500      500           351 Aug  7 14:51 exemple_execve.c
-rwxrwxr-x   1 500      500          4487 Aug  7 15:20 exemple_execvp
-rw-rw-r--   1 500      500           229 Aug  7 15:20 exemple_execvp.c
$ export PATH=/usr/bin
$ ./exemple_execvp
Erreur 2
$ export PATH=$PATH:/bin
$ ./exemple_execvp
total 12
-rwxrwxr-x   1 500      500          4607 Aug  7 14:53 exemple_execve
-rw-rw-r--   1 500      500           351 Aug  7 14:51 exemple_execve.c
-rwxrwxr-x   1 500      500          4487 Aug  7 15:20 exemple_execvp
-rw-rw-r--   1 500      500           229 Aug  7 15:20 exemple_execvp.c
$
```

La fonction `execlp()` permet de lancer une application qui sera recherchée dans les répertoires mentionnés dans la variable d'environnement `PATH`, en fournissant les arguments sous la forme d'une liste variable terminée par un pointeur `NULL`. Le prototype de **`execlp()`** est le suivant :

```
int execlp (const char * application, const char * arg, ...);
```

Cette présentation est plus facile à utiliser que `execvp()` lorsqu'on a un nombre précis d'arguments connus à l'avance. Si les arguments à transmettre sont définis dynamiquement durant le déroulement du programme, il est plus simple d'utiliser un tableau comme avec `execvp()`.

Voici un exemple de programme qui se rappelle lui-même en incrémentant un compteur transmis en argument. Il utilise `argv[0]` pour connaître son nom ; l'argument `argv[1]` contient alors le compteur qu'on incrémente jusqu'à 5 au maximum avant de relancer le même programme.

exemple_execlp.c

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (int argc, char * argv [])
{
    char compteur[2];
    int i;
```

```
    i = 0;
    if (argc == 2)
        sscanf(argv[1], "%d", & i);

    if (i < 5) {
        i ++;
        sprintf(compteur, "%d", i);

        fprintf(stdout, "execlp(%s, %s, %s, NULL)\n",
                        argv[0], argv[0], compteur);
        execlp(argv[0], argv[0], compteur, (char *) NULL);
    }
    return 0;
}
```

```
$ ./exemple_execlp
execlp(./exemple_execlp, ./exemple_execlp, 1, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 2, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 3, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 4, NULL)
execlp(./exemple_execlp, ./exemple_execlp, 5, NULL)
$
```

La fonction **`execl()`** est identique à `execlp()`, mais il faut indiquer le chemin d'accès complet, sans recherche dans `PATH`. La fonction **`execle()`** utilise le prototype suivant :

```
int execle(const char * app, const char * arg, ..., const char * envp []);
```

dans lequel on fournit un tableau explicite pour l'environnement désiré, comme avec `execve()`.

Récapitulons les caractéristiques des six fonctions de la famille `exec()`.

- `execv()`
  - tableau `argv[]` pour les arguments
  - variable externe globale pour l'environnement
  - nom d'application avec chemin d'accès complet
- `execve()`
  - tableau `argv[]` pour les arguments
  - tableau `envp[]` pour l'environnement
  - nom d'application avec chemin d'accès complet
- `execvp()`
  - tableau `argv[]` pour les arguments
  - variable externe globale pour l'environnement
  - application recherchée suivant le contenu de la variable `PATH`
- `execl()`
  - liste d'arguments `arg0, arg1, ..., NULL`
  - variable externe globale pour l'environnement
  - nom d'application avec chemin d'accès complet

- `execle()`
  - liste d'arguments `arg0, arg1, ..., NULL`
  - tableau `envp[]` pour l'environnement
  - nom d'application avec chemin d'accès complet
- `execlp()`
  - liste d'arguments `arg0, arg1, ..., NULL`
  - variable externe globale pour l'environnement
  - application recherchée suivant le contenu de la variable `PATH`

Lorsqu'un processus exécute un appel `exec()` et que celui-ci réussit, le nouveau programme remplace totalement l'ancien. Les segments de données, de code, de pile sont réinitialisés. En conséquence, les variables allouées en mémoire sont automatiquement libérées. Les chaînes d'environnement et d'argument sont copiées ; on peut donc utiliser n'importe quel genre de variables (statiques ou allouées dynamiquement, locales ou globales) pour transmettre les arguments de l'appel `exec()`.

L'ancien programme transmet automatiquement au nouveau programme :

- Les PID et PPID, PGID et SID. Il n'y a donc pas de création de nouveau processus.
- Les identifiants UID et GID, sauf si le nouveau programme est Set-UID ou Set-GID. Dans ce cas, seuls les UID ou GID réels sont conservés, les identifiants effectifs étant mis à jour.
- Le masque des signaux bloqués, et les signaux en attente.
- La liste des signaux ignorés. Un signal ayant un gestionnaire installé reprend son comportement par défaut. Nous discuterons de ce point dans le chapitre 7.
- Les descripteurs de fichiers ouverts ainsi que leurs éventuels verrous, sauf si le fichier dispose de l'attribut *close-on-exec* ; dans ce cas, il est refermé.

En revanche :

- Les temps d'exécution associés au processus ne sont pas remis à zéro.
- Les privilèges du nouveau programme dérivent des précédents comme nous l'avons décrit dans le chapitre 2.

Causes d'échec de lancement d'un programme

Nous avons dit que lorsque l'appel `exec()` réussit, il ne revient pas. Lorsque le programme lancé se finit par `exit()`, `abort()` ou `return` depuis la fonction `main()`, le processus est terminé. Par conséquent, lorsque `exec()` revient dans le processus appelant, une erreur s'est produite. Il est important d'analyser alors le contenu de la variable globale `errno` afin d'expliquer le problème à l'utilisateur. Le détail en est fourni dans la page de manuel de l'appel `exec()` considéré. Voyons les types d'erreurs pouvant se produire :

- Le fichier n'existe pas, n'est pas exécutable, le processus appelant n'a pas les autorisations nécessaires, ou l'interpréteur requis n'est pas accessible : `EACCES`, `EPERM`, `ENOEXEC`, `ENOENT`, `ENOTDIR`, `EINVAL`, `EISDIR`, `ELIBBAD`, `ENAMETOOLONG`, `ELOOP`. Le programme doit alors détailler l'erreur avant de proposer à l'utilisateur une nouvelle tentative d'exécution.
- Le fichier est trop gros, la mémoire manque, ou un problème d'ouverture de fichier se pose : `E2BIG`, `ENOMEM`, `EIO`, `ENFILE`, `EMFILE`. On peut considérer cela comme une erreur critique, où le programme doit s'arrêter, après avoir expliqué le problème à l'utilisateur.

- Un pointeur est invalide : `EFAULT`. Il s'agit d'un bogue de programmation.
- Le fichier est déjà ouvert en écriture : `ETXTBSY`.

Pour pouvoir détailler un peu cette dernière erreur, nous devons nous intéresser à la méthode employée par Linux pour gérer la mémoire virtuelle.

L'espace mémoire dont dispose un processus est découpé en pages. Ces pages mesurent 4 Ko sur les systèmes à base de 80x86, mais varient suivant les architectures des machines. Leur dimension est définie dans `<asm/param.h>`. Les processus ont l'impression d'avoir un espace d'adressage linéaire et continu, mais en réalité le noyau peut déplacer les pages à son gré dans la mémoire physique du système. Une collaboration entre le noyau et le processeur permet d'assurer automatiquement la traduction d'adresse nécessaire lors d'un accès mémoire.

Une page peut également ne pas se trouver en mémoire, mais résider sur le disque. Lorsque le processus tente d'y accéder, le processeur déclenche une *faute de page* et le noyau charge à ce moment la page désirée. Cela permet d'économiser la mémoire physique vraiment disponible.

Parallèlement, lorsque le noyau a besoin de trouver de la place en mémoire, il élimine une ou plusieurs pages qui ont peu de chances d'être utilisées dans un avenir proche. Si la page à supprimer a été modifiée par le processus, il est nécessaire de la sauvegarder sur le disque. Le noyau utilise alors la zone de *swap*. Si, au contraire, la page n'a pas été changée depuis son premier chargement sur le disque, on peut l'éliminer sans problème, le noyau sait où la retrouver.

Nous découvrons là une grande force de cette gestion mémoire : le code exécutable d'un programme, n'étant jamais modifié par le processus, n'a pas besoin d'être chargé entièrement en permanence. Le noyau peut relire sur le disque les pages de code nécessaires au fur et à mesure de l'exécution du programme. Il faut donc s'assurer qu'aucun autre processus ne risque de modifier le fichier exécutable. Pour cela, le noyau le verrouille, et toute tentative d'ouverture en écriture d'un fichier en cours d'exécution se soldera par un échec.

Un scénario classique pour un développeur met en avant ce phénomène : on utilise simultanément plusieurs consoles virtuelles ou plusieurs Xterm, en répartissant l'éditeur de texte sur une fenêtre, le compilateur sur une seconde, et le lancement du programme en cours de travail sur la troisième. Cela permet de relancer la compilation en utilisant simplement la touche de rappel de l'historique du shell, et de redémarrer le programme développé de la même manière dans une autre fenêtre. On apporte une modification au programme, et on oublie de le quitter avant de relancer la compilation. Le compilateur échouera alors en indiquant qu'il ne peut pas écrire sur un fichier exécutable en cours d'utilisation.

De la même façon, il n'est pas possible de lancer un programme dont le fichier est déjà ouvert en écriture par un autre processus. Dans ce cas, l'erreur `ETXTBSY` se produit. Il est bon dans ce cas de prévenir l'utilisateur. Le message peut même lui indiquer de se reporter à la commande `fuser` pour savoir quel processus a ouvert le fichier en question.

Nous allons mettre en lumière ce principe dans le programme suivant, `exemple_execv`, qui tente – vainement – d'ouvrir en écriture son propre fichier exécutable. Il ouvre ensuite en mode d'ajout en fin de fichier `exemple_execvp` que nous avons créé plus haut. Le fait d'ouvrir ce fichier en mode d'ajout évite de détruire les informations qu'il contient. Il tente alors de l'exécuter.

exemple_execv.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (int argc, char * argv[])
{
    int fd;

    char * nv_argv[] = { "./exemple_execvp", (char *) NULL };

    fprintf(stdout, "Essai d'ouverture de %s ... ", argv[0]);

    if ((fd = open(argv[0], O_WRONLY | O_APPEND)) < 0) {
        if (errno != ETXTBSY) {
            fprintf(stdout, "impossible, errno %d\n", errno);
            exit(1);
        }
        fprintf(stdout, "échec ETXTBSY, fichier déjà utilisé \n");
    }

    fprintf(stdout, "Ouverture de exemple_execvp en écriture ... ");

    if ((fd = open("exemple_execvp", O_WRONLY | O_APPEND)) < 0) {
        fprintf(stdout, "impossible, errno %d\n", errno);
        exit(1);
    }

    fprintf(stdout, "ok \n Tentative d'exécuter exemple_execvp ... ");
    execv("./exemple_execvp", nv_argv);

    if (errno == ETXTBSY)
        fprintf(stdout, "échec ETXTBSY fichier déjà utilisé \n");
    else
        fprintf(stdout, "errno = %d\n", errno);
    return 1;
}
```

Comme on pouvait s'y attendre, le programme n'arrive pas à ouvrir en écriture un fichier en cours d'exécution ni à lancer un programme dont le fichier est ouvert.

```
$ ls
exemple_execlp    exemple_execv    exemple_execve    exemple_execvp
exemple_execlp.c  exemple_execv.c  exemple_execve.c  exemple_execvp.c
$ ./exemple_execv
Essai d'ouverture ./exemple_execv ... échec ETXTBSY, fichier déjà utilisé
Ouverture de exemple_execvp en écriture ... ok
Tentative d'exécuter exemple_execvp ... échec ETXTBSY fichier déjà utilisé
$
```

Fonctions simplifiées pour exécuter un sous-programme

Il y a de nombreux cas où on désire lancer une commande externe au programme, sans pour autant remplacer le processus en cours. On peut par exemple avoir une application principale qui lance des sous-programmes indépendants, ou désirer faire appel à une commande système. Dans ce dernier cas, on peut classiquement invoquer la commande `mail` pour transmettre un message à l'utilisateur, à l'administrateur, ou envoyer un rapport de bogue au concepteur du programme.

Pour cela, nous disposons de la fonction `system()` et de la paire `popen()` / `pclose()`, qui sont implémentées dans la bibliothèque C en invoquant `fork()` et `exec()` selon les besoins.

La fonction **`system()`** est déclarée ainsi dans `<stdlib.h>` :

```
int system (const char * commande);
```

Cette fonction invoque le shell en lui transmettant la commande fournie, puis revient après la fin de l'exécution. Pour ce faire, il faut exécuter un `fork()`, puis le processus lance la commande en appelant le shell « `/bin/sh -c commande` », tandis que le processus père attend la fin de son fils. Si l'invocation du shell échoue, `system()` renvoie 127. Si une autre erreur se produit, elle renvoie –1, sinon elle renvoie la valeur de retour de la commande exécutée. Une manière simplifiée d'implémenter `system()` pourrait être la suivante :

```
    int
notre_system (const char * commande)
{
    char *   argv[4];
    int      retour;
    pid_t    pid;

    if ((pid = fork()) < 0)
        /* erreur dans fork */
        return -1;

    if (pid == 0) {
        /* processus fils */
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = commande;
        argv[3] = (char *) NULL;
        execv("/bin/sh", argv);
        /* execv a échoué */
        exit(127);
    }

    /* processus père */
    /* attente de la fin du processus fils */
    while (waitpid(pid, & retour, 0) < 0)
        if (errno != EINTR)
            return -1;
    return retour;
}
```

Attention
La fonction `system()` représente une énorme faille de sécurité dans toute application installée Set-UID. Voyons le programme simple suivant :

exemple_system.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    system("ls");
    return 0;
}
```

Le programme ne fait que demander au shell d'exécuter "`ls`". Pourtant, si on l'installe Set-UID *root*, il s'agit d'une faille de sécurité. En effet, lorsque le shell recherche la commande "`ls`", il parcourt les répertoires mentionnés dans la variable d'environnement `PATH`. Celle-ci est héritée du processus père et peut donc être configurée par l'utilisateur pour inclure en premier le répertoire ".". Le shell exécutera alors de préférence la commande "`ls`" qui se trouve dans le répertoire en cours. Il suffit que l'utilisateur crée un shell script exécutable, et le tour est joué. Voyons un exemple, en créant le shell script suivant :

ls :

```
#! /bin/sh
echo faux ls
echo qui lance un shell
sh
```

Examinons l'exécution suivante :

```
$ ./exemple_system
exemple_execlp    exemple_execv.c   exemple_execvp    exemple_system.c
exemple_execlp.c  exemple_execve    exemple_execvp.c  ls
exemple_execv     exemple_execve.c  exemple_system
$ export PATH=.:$PATH
$ ./exemple_system
faux ls
qui lance un shell
$ exit
$
```

Tout d'abord, le programme s'exécute normalement et invoque « `sh -c ls` », qui trouve `ls` dans le répertoire `/bin` comme d'habitude. Ensuite, nous modifions notre `PATH` pour y placer en première position le répertoire en cours. À ce moment, le shell exécutera notre "`ls`" piégé qui lance un shell. Jusque-là, rien d'inquiétant. Mais imaginons maintenant que le programme soit Set-UID *root*. C'est ce que nous configurons avant de revenir en utilisateur normal.

```
$ su
Password:
# chown root.root exemple_system
```

```
# chmod +s exemple_system
# exit
$
```

À ce moment, l'exécution du programme lance le "`ls`" piégé avec l'identité de *root* !

```
$ ./exemple_system
faux ls
qui lance un shell
#
```

Comme nous avons inclus dans notre script une invocation de shell, nous nous retrouvons avec un shell connecté sous *root* ! Il ne faut pas s'imaginer que le fait de forcer la variable d'environnement `PATH` dans le programme aurait résolu le problème. D'autres failles de sécurité classiques existent, notamment en faussant la variable d'environnement `IFS` qui permet au shell de séparer ses arguments (normalement des espaces, des tabulations, etc.).

Il ne faut donc jamais employer la fonction `system()` dans un programme Set-UID (ou Set-GID). On peut utiliser à la place les fonctions `exec()`, qui ne parcourent pas les répertoires du `PATH`. Le vrai danger avec `system()` est qu'il appelle le shell au lieu de lancer la commande directement.

La véritable version de `system()`, présente dans la GlibC, est légèrement plus complexe puisqu'elle gère les signaux `SIGINT` et `SIGQUIT` (en les ignorant) et `SIGCHLD` (en le bloquant).

En théorie, le fait de transmettre une commande `NULL` sert à vérifier la présence du shell `/bin/sh`. Normalement, `system()` doit renvoyer une valeur non nulle s'il est bien là. En pratique, sous Linux, la vérification n'a pas lieu, GlibC considère que `/bin/sh` appartient au minimum vital d'un système Unix.

Après avoir bien compris que la fonction `system()` ne doit jamais être employée dans un programme Set-UID ou Set-GID, rien n'empêche de l'utiliser dans des applications simples ne nécessitant pas de privilèges. L'exemple que nous invoquions précédemment concernant l'appel de l'utilitaire `mail` est pourtant difficile à utiliser avec la fonction `system()`, car il faudrait d'abord créer un fichier contenant le message, puis lancer `mail` avec une redirection d'entrée.

Pour cela, il est plus pratique d'utiliser la fonction **`popen()`**, qui permet de lancer un programme à la manière de `system()`, mais en fournissant un des flux d'entrée ou de sortie standard pour dialoguer avec le programme appelant.

Le prototype de cette fonction, dans `<stdio.h>`, est le suivant :

```
FILE * popen (const char * commande, const char * mode);
```

La commande est exécutée comme avec `system()` en invoquant `fork()` et `exec()`, mais, de plus, le flux d'entrée ou de sortie standard de la commande est renvoyé au processus appelant. La chaîne de caractères `mode` doit contenir soit `r` (*read*), si on souhaite lire les données de la sortie standard de la commande dans le flux renvoyé, `w` (*write*) si on préfère écrire sur son entrée standard. Le flux renvoyé par la fonction `popen()` est tout à fait compatible avec les fonctions d'entrée-sortie classiques telles `fprintf()`, `fscanf()`, `fread()` ou `fwrite()`. Par contre, le flux doit toujours être refermé en utilisant la fonction `pclose()` à la place de `fclose()`.

Lorsqu'on appelle **`pclose()`**, cette fonction attend que le processus exécutant la commande se termine, puis renvoie son code de retour.

Voici un exemple simple dans lequel nous avons exécuté la commande "`mail`", suivie de notre nom d'utilisateur obtenu avec `getlogin()`. La commande est exécutée en redirigeant son flux d'entrée standard. Nous pouvons donc écrire notre message tranquillement par une série de `fprintf()`.

exemple_popen_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    FILE * message;
    char * commande;

    if ((commande = malloc(strlen(getlogin()) + 6)) == NULL) {
        fprintf(stderr, "Erreur malloc %d\n", errno);
        exit(1);
    }

    strcpy(commande, "mail ");
    strcat(commande, getlogin());

    if ((message = popen(commande, "w")) == NULL) {
        fprintf(stderr, " Erreur popen %d \n", errno);
        exit(1);
    }

    fprintf(message, "Ceci est un message \n");
    fprintf(message, "émis par moi-meme\n");

    pclose(message);

    return 0;
}
```

Lorsqu'il est lancé, ce programme émet bien le mail prévu. On notera que `popen()` effectue, comme `system()`, un `execl()` de `/bin/sh -c commande`. Cette fonction est donc recherchée dans les répertoires mentionnés dans le `PATH`.

Une autre application classique de `popen()`, utilisant l'entrée standard de la commande exécutée, est d'invoquer le programme indiqué dans la variable d'environnement `PAGER`, ou si elle n'existe pas, `less` ou `more`. Ces utilitaires affichent les données qu'on leur envoie page par page, en s'occupant de gérer la taille de l'écran (`less` permet même de revenir en arrière). C'est un moyen simple et élégant de fournir beaucoup de texte à l'utilisateur en lui laissant la possibilité de le consulter à sa guise.

Notre second exemple va lire la sortie standard de la commande exécutée. C'est une méthode généralement utilisée pour récupérer les résultats d'une application complémentaire ou pour

invoquer une commande système qui fournit des données difficiles à obtenir directement (`who`, `ps`, `last`, `netstat`…).

Nous allons ici invoquer la commande `ifconfig` en lui demandant l'état de l'interface réseau `eth0`. Si celle-ci est activée, `ifconfig` renvoie une sortie du genre :

```
eth0 Lien encap:Ethernet  HWaddr 00:50:04:8C:7A:ED
     inet adr:172.16.15.16  Bcast:172.16.255.255  Masque:255.255.0.0
     UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
     Paquets Reçus:0 erreurs:0 jetés:0 débordements:7395 trames:0
     Paquets transmis:29667 erreurs:0 jetés:0 débordements:0 carrier:22185
     collisions:7395 lg file transmission:100
     Interruption:3 Adresse de base:0x200
```

Si `eth0` est désactivée, on obtient :

```
eth0 Lien encap:Ethernet  HWaddr 00:50:04:8C:7A:ED
     inet adr:172.16.15.16  Bcast:172.16.255.255  Masque:255.255.0.0
     BROADCAST MULTICAST  MTU:1500  Metric:1
     Paquets Reçus:0 erreurs:0 jetés:0 débordements:11730 trames:0
     Paquets transmis:47058 erreurs:0 jetés:0 débordements:0 carrier:35190
     collisions:11730 lg file transmission:100
     Interruption:3 Adresse de base:0x200
```

(Remarquez la différence dans la troisième ligne, `UP` dans un cas, et pas dans l'autre.) Si l'interface n'existe pas, `ifconfig` ne renvoie rien sur sa sortie standard, mais écrit un message :

```
eth0: erreur lors de la recherche d'infos sur l'interface: Périphérique non trouvé
```

sur sa sortie d'erreur.

Notre programme va donc lancer la commande et rechercher si une ligne de la sortie standard commence par `UP`. Si c'est le cas, il indique que l'interface est active. S'il ne trouve pas cette chaîne de caractères ou si la commande ne renvoie aucune donnée sur sa sortie standard, il considère l'interface comme étant inactive.

exemple_popen_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <errno.h>

    int
main (void)
{
    FILE * sortie;
    char ligne [128];
    char etat  [128];

    if ((sortie = popen("/sbin/ifconfig eth0", "r")) == NULL) {
        fprintf(stderr, " Erreur popen %d \n", errno);
        exit(1);
    }
```

```
    while (fgets(ligne, 127, sortie) != NULL) {
        if (sscanf(ligne, "%s", etat) == 1)
            if (strcmp(etat, "UP") == 0) {
                fprintf(stderr, "interface eth0 en marche \n");
                pclose(sortie);
                return 0;
            }
    }
    fprintf(stdout, "interface eth0 inactive \n");
    pclose(sortie);
    return 0;
}
```

Cet exemple (un peu artificiel, convenons-en) montre quand même l'utilité d'invoquer une commande système et d'en récupérer aisément les informations. Encore une fois, insistons sur le manque de sécurité qu'offre `popen()` pour un programme susceptible d'être installé Set-UID ou Set-GID.

Un dernier exemple concernant `popen()` nous permet d'invoquer un script associé à l'application principale. Ce script est écrit en langage Tcl/Tk, et offre une boîte de saisie configurable. Il utilise les chaînes de caractères transmises en argument en ligne de commande :

- Le premier argument correspond au nom de la boîte de saisie (le titre de la fenêtre).
- Le second argument est le libellé affiché pour questionner l'utilisateur.
- Le troisième argument (éventuel) est la valeur par défaut pour la zone de saisie.

En invoquant ainsi ce script dans un Xterm :

```
$ ./exemple_popen_3.tk Approximation "Entrez le degré du polynôme pour
 l'approximation des trajectoires" 3
```

La fenêtre suivante apparaît :

Figure 4.1
Fenêtre de saisie en Tcl/Tk

Lorsqu'on appuie sur le bouton Ok, la valeur saisie est affichée sur la sortie standard.

Le script Tcl/Tk est volontairement simplifié ; il ne traite aucun cas d'erreur.

exemple_popen_3.tk :

```
#! /usr/bin/wish

## Le titre de la fenêtre est le premier argument reçu
## sur la ligne de commande.
wm title . [lindex $argv 0]
```

```
## Le haut de la boîte de dialogue contient un libellé
## fourni en second argument de la ligne de commande, et
## une zone de saisie dont le contenu par défaut est
## éventuellement fourni en troisième argument.
frame .haut -relief flat -borderwidth 2
label .libelle -text [lindex $argv 1]
entry .saisie -relief sunken -borderwidth 2
.saisie insert 0 [lindex $argv 2]
pack .libelle .saisie -in .haut -expand true -fill x

## Le bas contient deux boutons, Ok et Annuler, chacun avec
## sa procédure associée.
frame .bts -relief sunken -borderwidth 2
button .ok -text "Ok" -command bouton_ok
button .annuler -text "Annuler" -command bouton_annuler
pack .ok .annuler -side left -expand true -pady 3 -in .bts
pack .haut .bts
update

proc bouton_ok {} {
    ## La procédure associée à OK transmet la chaîne lue
    ## sur la sortie standard.
    puts [.saisie get]
    exit 0
}

proc bouton_annuler {} {
    ## Si on annule, on n'écrit rien sur la sortie standard.
    ## On quitte simplement.
    exit 0
}
```

Notre programme C va invoquer le script et traiter quelques cas d'échec, notamment en testant le code de retour de `pclose()`. Si une erreur se produit, on effectue la saisie à partir de l'entrée standard du processus. Ceci permet d'utiliser le même programme dans un environnement X-Window avec une boîte de dialogue ou sur une console texte avec une saisie classique.

La ligne de commande que `popen()` invoque est la suivante :

```
./exemple_popen_3.tk Saisie "Entrez votre nom" nom_login 2> /dev/null
```

dans laquelle *nom_login* est obtenu par la commande `getlogin()`.

On redirige la sortie d'erreur standard vers `/dev/null` afin d'éviter les éventuels messages d'erreur de Tk si on se trouve sur une console texte (on suppose que le shell `/bin/sh` utilisé par `popen()` est du type Bourne, ce qui est normalement le cas sous Linux). La chaîne "Entrez votre nom" est encadrée par des guillemets pour qu'elle ne constitue qu'un seul argument de la ligne de commande.

Voici le programme C qui invoque le script décrit précédemment :

exemple_popen_3.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>

    int
main (void)
{
    FILE * saisie;
    char * login ;
    char nom [128];
    char commande [128];

    if ((login = getlogin()) == NULL)
        strcpy(nom, "\"\"") ;
    else
        strcpy(nom, login) ;
    sprintf(commande, "./exemple_popen_3.tk "
                      "Saisie "
                      "\"Entrez votre nom\" "
                      "%s 2>/dev/null", nom);

    if ((saisie = popen(commande , "r")) == NULL) {
        /* Le script est, par exemple, introuvable */
        /* On va essayer de lire sur stdin.        */
        fprintf(stdout, "Entrez votre nom : ");
        if (fscanf(stdin, "%s", nom) != 1) {
            /* La lecture sur stdin échoue...         */
            /* On utilise une valeur par défaut.      */
            strcpy(nom, getlogin());
        }
        fprintf(stdout, "Nom saisi : %s\n", nom);
        return 0;
    }

    if (fscanf(saisie, "%s", nom) != 1) {
        if (pclose(saisie) != 0) {
            /* Le script a échoué pour une raison quelconque. */
            /* On recommence la saisie sur stdin.             */
            fprintf(stdout, "Entrez votre nom : ");
            if (fscanf(stdin, "%s", nom) != 1) {
                /* La lecture sur stdin échoue...         */
                /* On utilise une valeur par défaut.      */
                strcpy(nom, getlogin());
            }
        } else {
            /* L'utilisateur a cliqué sur Annuler. Il faut    */
            /* abandonner l'opération en cours.               */
```

```
            fprintf(stdout, "Pas de nom fourni - abandon\n");
            return 1;
        }
    } else {
        pclose(saisie);
    }
    fprintf(stdout, "Nom saisi : %s\n", nom);
    return 0;
}
```

Conclusion

Ce chapitre nous a permis de découvrir plusieurs méthodes pour lancer une application. Les mécanismes à base de `exec()` permettent de remplacer totalement le programme en cours par un autre qui est exécutable, tandis que les fonctions `system()` et `popen()`-`pclose()` servent plutôt à utiliser une autre application comme sous-programme de la première.

5

Fin d'un programme

Dans ce chapitre, nous allons étudier tout d'abord les moyens de mettre fin à l'exécution d'un programme. Nous verrons ensuite des méthodes permettant d'enregistrer des routines qui seront automatiquement exécutées avant de quitter l'application.

Nous nous pencherons sur l'attente de la fin d'un processus fils et la récupération de son état de terminaison, puis nous examinerons les moyens de signaler une erreur à l'utilisateur, même si celle-ci ne conduit pas nécessairement à l'arrêt du programme.

Terminaison d'un programme

Un processus peut se terminer normalement ou anormalement. Dans le premier cas, l'application est abandonnée à la demande de l'utilisateur, ou la tâche à accomplir est finie. Dans le second cas, un dysfonctionnement est découvert, qui est si sérieux qu'il ne permet pas au programme de continuer son travail. Le processus est alors tué par le noyau par l'intermédiaire d'un signal fatal.

Terminaison normale d'un processus

Un programme peut se finir de plusieurs manières. La plus simple est de revenir de la fonction `main()` en renvoyant un compte rendu d'exécution sous forme de valeur entière. Cette valeur est lue par le processus père, qui peut en tirer les conséquences adéquates. Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle, tandis que les cas d'échec sont indiqués par des codes de retour non nuls (et qui peuvent être documentés avec l'application). Cela permet d'écrire des scripts shell robustes, qui vérifient le bon fonctionnement de chaque commande employée. Dans la plupart des cas, on ne teste que la nullité du code de retour. Lorsque le processus est arrêté à cause d'un signal, le shell modifie le code de retour (*bash* ajoute 128, par exemple). Il est donc conseillé de n'utiliser que des valeurs comprises entre 0 et 127.

Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale `$?`. Suivant les applications, ce code aura différentes significations. Prenons par exemple l'utilitaire `grep`. Nous allons l'invoquer de manière à ce qu'il cherche – et trouve – la chaîne de caractères `root` dans le fichier `/etc/passwd`.

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$ echo $?
0
$
```

Le code de retour est nul, indiquant la réussite. À présent, donnons lui à chercher une chaîne ne se trouvant pas dans le fichier :

```
$ grep abcdefg /etc/passwd
$ echo $?
1
$
```

La valeur `1` signifie donc *« j'ai fait mon travail correctement, mais je n'ai pas trouvé la chaîne »*. Demandons-lui maintenant de consulter un fichier inexistant :

```
$ grep root /etc/inexistant
grep: /etc/inexistant: Aucun fichier ou répertoire de ce type
$ echo $?
2
$
```

Le code de retour `2` a donc une signification différente : « *je n'ai pas pu faire mon travail, la demande est invalide* ». Tous ces codes (documentés dans la page de manuel de `grep`), sont renvoyés par le processus quand il se termine normalement – de son plein gré. Toutefois, si le processus est interrompu prématurément et se termine anormalement, le shell renseigne la variable spéciale `$?` en fonction du numéro du signal ayant tué le processus. Par exemple, nous invoquons la commande `sleep` pour un sommeil de 30 secondes, et nous l'interrompons pendant ce temps en pressant Contrôle-C :

```
$ sleep 30
     (Ctrl-C)
$ echo $$
130
$
```

La valeur `130` correspond à `128+2`. Comme on le voit dans la page de manuel `signal(7)`, ce numéro est celui du signal `SIGINT`, correspondant à la pression de la touche d'interruption (Ctrl-C par défaut).

Si seuls la réussite ou l'échec du programme importent (si le processus père n'essaye pas de détailler les raisons de l'échec), il est possible d'employer les constantes symboliques `EXIT_SUCCESS` ou `EXIT_FAILURE` définies dans `<stdlib.h>`. Ceci a l'avantage d'adapter automatiquement le comportement du programme, même sur les systèmes non conformes à SUSv3, où ces constantes ne sont pas nécessairement 0 et 1.

Une autre façon de terminer un programme normalement est d'utiliser la fonction **exit()**.

```
void exit (int code);
```

On lui transmet en argument le code de retour pour le processus père. L'effet est strictement égal à celui d'un retour depuis la fonction `main()`, à la différence que `exit()` peut être invoquée depuis n'importe quelle partie du programme (notamment depuis les routines de traitement d'erreur).

Lorsqu'on utilise uniquement une terminaison avec `exit()` dans un programme, le compilateur se plaint que la fin de la fonction `main()` est atteinte alors qu'aucune valeur n'a été renvoyée.

exemple_exit_1.c :

```
    #include <stdlib.h>

    void sortie (void);

    int
main (void)
{
    sortie();
}

    void
sortie(void)
{
    exit(EXIT_FAILURE);
}
```

déclenche à la compilation l'avertissement suivant :

```
$ cc -Wall exemple_exit_1.c -o exemple_exit_1
exemple_exit_1.c: In function `main':
exemple_exit_1.c:9: warning: control reaches end of non-void function
$
```

(Si nous avions directement mis `exit()` dans la fonction `main()`, le compilateur l'aurait reconnu et aurait supprimé cet avertissement.)

Pour éviter ce message, on peut être tenté de déclarer `main()` comme une fonction de type `void`. Sous Linux, cela ne pose pas de problème, mais un tel programme pourrait ne pas être portable sur d'autres systèmes qui exigent que `main()` renvoie une valeur. D'ailleurs, le compilateur `gcc` avertit que `main()` doit normalement être de type `int`.

exemple_exit_2.c :

```
    #include <stdlib.h>

    void
main (void)
{
    exit(EXIT_SUCCESS);
}
```

déclenche un avertissement :

```
$ cc -Wall exemple_exit_2.c -o exemple_exit_2
exemple_exit_2.c:5: warning: return type of `main' is not `int'
$
```

Ayons donc comme règle de bonne conduite – ou plutôt de bonne lisibilité – de toujours déclarer `main()` comme étant de type `int`, et ajoutons systématiquement un `return 0` ou `return EXIT_SUCCESS` à la fin de cette routine. C'est une bonne habitude à prendre, même si nous sortons toujours du programme en invoquant `exit()`.

Lorsqu'un processus se termine normalement, en revenant de `main()` ou en invoquant `exit()`, la bibliothèque C effectue les opérations suivantes :

- Elle appelle toutes les fonctions qui ont été enregistrées à l'aide des routines `atexit()` et `on_exit()`, que nous verrons dans la prochaine section.
- Elle ferme tous les flux d'entrée-sortie, en écrivant effectivement toutes les données qui étaient en attente dans les buffers.
- Elle supprime les fichiers créés par la fonction `tmpfile()`.
- Elle invoque l'appel-système `_exit()` qui terminera le processus.

L'appel-système **`_exit()`** exécute – pour ce qui concerne le programmeur applicatif – les tâches suivantes :

- Il ferme les descripteurs de fichiers (transférant les données aux périphériques).
- Les processus fils sont adoptés par le processus 1 (`init`), qui lira leur code de retour dès qu'ils se finiront pour éviter qu'ils ne restent à l'état zombie de manière prolongée.
- Le processus père reçoit un signal `SIGCHLD`.
- Selon certaines conditions et si le processus est leader de session, le signal `SIGHUP` peut être envoyé à tous les processus en avant-plan sur le terminal de la session.
- Si le processus est leader de son groupe et s'il y a des processus stoppés dans celui-ci, tous les membres du groupe à présent orphelins reçoivent `SIGHUP` et `SIGCONT`.

Le système se livre également à des tâches de libération des ressources verrouillées, de comptabilisation éventuelle des processus, etc. Le détail de ces opérations n'est pas d'une grande importance pour une application classique, considérons simplement que l'exécution du processus est terminée, et que ses ressources sont libérées.

Le processus devient alors un zombie, c'est-à-dire qu'il attend que son processus père lise son code de retour. Si le processus père ignore explicitement `SIGCHLD`, le noyau effectue automatiquement cette lecture. Si le processus père s'est déjà terminé, `init` adopte temporairement le zombie, juste le temps de lire son code de retour. Une fois cette lecture effectuée, le processus est éliminé de la liste des tâches sur le système.

Le fait d'invoquer `_exit()` à la place de `exit()` peut être utile dans certaines circonstances :

- Lorsqu'on utilise un partage des fichiers entre le processus père et le processus fils, par exemple en employant `__clone()` à la place de `fork()`, ou lors de l'implémentation d'une bibliothèque de *threads*. À ce moment-là, on évite de fermer les flux d'entrée-sortie, car le processus père peut encore avoir besoin de ces fichiers. Ce cas est assez rare dans des applications courantes.

- On peut enregistrer des routines pour qu'elles soient automatiquement exécutées lors de la sortie du programme par `exit()` ou `return()` depuis `main()`. Ces fonctions servent généralement à faire du « ménage » ou à signaler explicitement la fin d'une transaction sur une connexion réseau. Le fait d'appeler directement `_exit()` empêchera l'exécution de ces routines.

Si on utilise `_exit()`, il ne faut pas oublier de fermer proprement tous les fichiers pour être sûr que les données temporairement en buffer soient écrites entièrement. De même, les éventuels fichiers temporaires créés par `tmpfile()` ne sont pas détruits automatiquement.

Terminaison anormale d'un processus

Un programme peut également se terminer de manière anormale. Ceci est le cas par exemple lorsqu'un processus exécute une instruction illégale, ou qu'il essaye d'accéder au contenu d'un pointeur mal initialisé. Ces actions déclenchent un signal qui, par défaut, arrête le processus en créant un fichier d'image mémoire `core`. Nous en parlerons plus longuement dans le prochain chapitre.

Une manière « propre » d'interrompre anormalement un programme (par exemple lorsqu'un bogue est découvert) est d'invoquer la fonction **`abort()`**.

```
void abort (void);
```

Celle-ci envoie immédiatement au processus le signal `SIGABRT`, en le débloquant s'il le faut, et en rétablissant le comportement par défaut si le signal est ignoré. Nous verrons dans le prochain chapitre la manière de traiter ce signal si on désire installer un gestionnaire pour effectuer quelques tâches de nettoyage avant de finir le programme.

Le problème de la fonction `abort()` ou des arrêts dus à des signaux est qu'il est difficile de déterminer ensuite à quel endroit du programme le dysfonctionnement a eu lieu. Il est possible d'autopsier le fichier `core` (à condition d'avoir inclus les informations de débogage lors de la compilation avec l'option `-g` de `gcc`), mais c'est une tâche parfois ardue. Une autre manière de détecter automatiquement les bogues est d'utiliser systématiquement la fonction **`assert()`** dans les parties critiques du programme. Il s'agit d'une macro, définie dans `<assert.h>`, et qui évalue l'expression qu'on lui transmet en argument. Si l'expression est vraie, elle ne fait rien. Par contre, si elle est fausse, `assert()` arrête le programme après avoir écrit un message sur la sortie d'erreur standard, indiquant le fichier source concerné, la ligne de code et le texte de l'assertion ayant échoué. Il est alors très facile de se reporter au point décrit pour rechercher le bogue.

La macro `assert()` agit en surveillant perpétuellement que les conditions prévues pour l'exécution du code soient respectées. Voici un exemple où nous faisons volontairement échouer la seconde assertion.

exemple_assert.c :

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void fonction_reussissant (int i);
void fonction_echouant    (int i);
```

```
    int
main (void)
{
    fonction_reussissant(5);
    fonction_echouant(5);

    return EXIT_SUCCESS;
}

    void
fonction_reussissant (int i)
{
    /* Cette fonction nécessite que i soit positif */

    assert(i >= 0);
    fprintf(stdout, "Ok, i est positif \n");
}

    void
fonction_echouant (int i)
{
    /* Cette fonction nécessite que i soit négatif */

    assert(i <= 0);
    fprintf(stdout, "Ok, i est négatif \n");
}
```

Lors de l'exécution, la première assertion passe, et le message est écrit sur `stdout`, mais la seconde assertion échoue, et `assert()` affiche alors le détail du problème sur `stderr` :

```
$ ./exemple_assert
Ok, i est positif
exemple_assert: exemple_assert.c:31: fonction_echouant:  l'assertion `i <= 0'
➥ a échoué.
Aborted (core dumped)
$
```

Nous voyons alors le grand intérêt de `assert()` : elle nous indique le nom du programme exécutable, le fichier source concerné, le numéro de la ligne, le nom de la fonction, et le texte intégral de l'assertion ayant engendré l'arrêt. De plus, elle déclenche la création d'un fichier `core` pouvant servir à analyser plus en détail l'état des données au moment de l'échec.

Notre exemple est assez artificiel car nous avons utilisé une macro `assert()` pour vérifier des conditions qui auraient très bien pu être analysées par une structure `if/else` renvoyant un code d'erreur. En fait, `assert()` ne doit être utilisée que pour des circonstances ne devant jamais se produire durant l'exécution normale du programme.

En effet, lorsque la phase de débogage est terminée, on supprime toutes les assertions en définissant une constante symbolique spéciale (`NDEBUG`) à la compilation. Cela permet de gagner en efficacité en éliminant tous ces tests qui sont dorénavant inutiles.

Attention

`assert()` étant définie comme une macro qui évalue son argument, il faut éviter totalement tous les effets de bord dans l'expression transmise. On n'utilisera donc jamais des formulations du genre : `assert(++i < 5)` ou `assert((i = j) != 0)`, car la partie assignation de ces expressions disparaît lorsqu'on passe en code de production.

Il est bon d'utiliser systématiquement `assert()` pour vérifier les arguments d'entrée d'une routine lorsqu'ils doivent, dans tous les cas, se situer dans une plage de valeurs données (taille supérieure à 0, pointeur non `NULL`…). Ceci permet d'ailleurs de documenter automatiquement les contraintes sur les arguments attendus, une ligne :

```
assert(ptr != NULL);
```

étant aussi parlante et plus efficace qu'un commentaire

```
/* On suppose que le pointeur n'est jamais NULL */
```

qui risque de ne pas être mis à jour en cas de modification du code.

On notera qu'il est possible de définir avec `#define` ou de supprimer avec `#undef` la constante `NDEBUG` dans le corps même d'un module, en ré-incluant `<assert.h>` à la suite. La macro `assert()` sera lors validée ou ignorée jusqu'à la prochaine modification. Ceci permet de n'activer le débogage que dans des portions restreintes du logiciel, ou au contraire d'exclure des fonctions qui ont été totalement validées.

Face à un test précis, il est parfois difficile de décider s'il faut l'incorporer dans une assertion ou dans une vérification plus classique avec un message sur `stderr`. La règle est que le code définitif qui sera soumis à l'utilisateur ne pourra en aucun cas faire échouer une assertion. La fonction `assert()` est un outil de débogage et pas une méthode de sortie sur erreur. C'est pourquoi, si une demande d'allocation mémoire échoue, la gestion d'erreur doit être effectuée par le programme directement, car ici ce n'est pas un bogue mais un problème de ressource indisponible temporairement. Cependant, une routine de traitement de chaîne de caractères peut refuser systématiquement un pointeur `NULL` en entrée. Dans ce cas, une assertion échouant permet de rechercher (à l'aide du fichier `core` créé) la routine appelante ayant transmis un mauvais argument.

Lorsqu'on désire basculer en code définitif pour l'utilisateur, on peut inclure à la compilation la constante `NDEBUG` sur la ligne de commande de `gcc` :

```
$ cc -Wall -DNDEBUG programme.c -o programme
```

ou dans le corps du fichier C, avant l'inclusion de `<assert.h>` :

```
#define NDEBUG
#include <assert.h>
```

Cette dernière méthode permet de faire basculer indépendamment en mode de développement ou de production les divers modules du projet.

Notons finalement que `assert()` est implémentée dans la GlibC en appelant `abort()`, ce qui signale une terminaison anormale du processus père, comme nous en verrons le détail dans les routines `wait()`, `waitpid()`, etc.

Exécution automatique de routines de terminaison

Il est possible, grâce aux routines `atexit()` et `on_exit()` de faire enregistrer des fonctions qui seront automatiquement invoquées lorsque le processus se terminera normalement, c'est-à-dire par un retour de la fonction `main()` ou par un appel de la fonction `exit()`.

Ces routines peuvent être utiles dans plusieurs cas :

- effacer des fichiers temporaires ou au contraire enregistrer les préférences de l'utilisateur, ou l'historique des actions effectuées ;
- enregistrer sur disque les structures d'une base de données maintenue en mémoire ;
- libérer les verrous sur les fichiers ou bases de données partagés ;
- signaler la fin du processus au démon de journalisation du système (`syslog`) ;
- restaurer l'état initial du terminal ;
- terminer un dialogue réseau proprement en suivant un protocole complet, plutôt que de simplement couper la connexion…

Il est toujours possible d'appeler explicitement ces routines avant de quitter l'application, mais l'avantage de ce mécanisme d'invocation automatique est double :

- On peut quitter le programme depuis plusieurs endroits en appelant `exit()`, ou revenir de la fonction `main()` sans avoir à se soucier des routines de terminaison. Elles seront appelées systématiquement quel que soit le cas.
- Lorsqu'on définit une bibliothèque de fonctions pouvant être réutilisées dans plusieurs programmes, et que celle-ci nécessite un traitement final avant la fin du processus (par exemple pour l'un des cas cités ci-dessus), il est plus sûr d'enregistrer avec `atexit()` la routine désirée plutôt que de demander au programmeur qui utilisera la bibliothèque d'appeler une fonction finale.

Le prototype de **`atexit()`** est déclaré dans `<stdlib.h>`, ainsi :

```
int atexit (void * routine (void));
```

En d'autres termes, on doit lui transmettre un pointeur sur une routine de type :

```
void routine_terminaison (void);
```

Lorsqu'elle réussit, `atexit()` renvoie 0. Sinon, elle renvoie une valeur non nulle. La norme C Ansi indique qu'on peut enregistrer au minimum 32 fonctions. La GlibC ne fixe pas de limites, en allouant dynamiquement les structures de données nécessaires à la mémorisation.

Les fonctions mémorisées avec `atexit()` sont invoquées, en sortie, dans l'ordre inverse de leur enregistrement. Une fonction enregistrée deux fois est invoquée deux fois. Il n'y a pas de possibilité de « déprogrammer » une fonction mémorisée. La meilleure solution pour désactiver une routine de terminaison est d'utiliser une variable globale que la routine consultera pour savoir si elle doit agir ou non.

Lorsqu'on appelle la fonction `exit()` depuis l'intérieur d'une routine de terminaison, elle n'a pas d'effet (en particulier, le programme ne boucle pas sur cette routine). Par contre, si on invoque l'appel-système `_exit()`, la sortie est immédiate, sans appeler les éventuelles autres routines de terminaison. Les routines de terminaison sont invoquées avant la fermeture systématique des fichiers ouverts et l'effacement des fichiers temporaires fournis par `tmpfile()`. Il est donc possible de les utiliser encore dans les routines de terminaison.

Voici un exemple d'utilisation de la fonction `atexit()`. Nous appellerons trois routines, et la deuxième sera enregistrée deux fois. On voit également que l'appel-système `exit()` n'a pas d'effet lorsqu'il est appelé depuis l'intérieur d'une routine de terminaison.

exemple_atexit.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    void sortie_1 (void);
    void sortie_2 (void);
    void sortie_3 (void);

    int
main (void)
{
    if (atexit(sortie_3) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_3()\n");
    if (atexit(sortie_2) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_2()\n");
    if (atexit(sortie_2) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_2()\n");
    if (atexit(sortie_1) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_1()\n");
    fprintf(stdout, "Allez... on quitte en revenant de main()\n");
    return EXIT_SUCCESS;
}

    void
sortie_1 (void)
{
    fprintf(stdout, "Sortie_1 : appelle exit()\n");
    exit(EXIT_SUCCESS);
}

    void
sortie_2 (void)
{
    fprintf(stdout, "Sortie_2\n");
}

    void
sortie_3 (void)
{
    fprintf(stdout, "Sortie_3\n");
}
```

L'exécution a lieu ainsi :

```
$ ./exemple_atexit
Allez... on quitte en revenant de main()
Sortie_1 : appelle exit()
```

```
Sortie_2
Sortie_2
Sortie_3
$
```

Par contre, si on remplace `exit()` par `_exit()` dans `sortie_1()`, on obtient :

```
$ ./exemple_atexit
Allez... on quitte en revenant de main()
Sortie_1 : appelle _exit()
$
```

Il existe une seconde fonction permettant d'enregistrer des routines de terminaison : **`on_exit()`**. Il s'agit d'une extension Gnu. La routine de terminaison recevra lors de son invocation deux arguments : le premier est un entier correspondant au code transmis à `exit()`ou `return` de `main()`. Le second argument est un pointeur `void *`, et la valeur de cet argument est programmée lors de l'appel de `on_exit()`.

Le prototype de `on_exit()` est déclaré dans `<stdlib.h>`, ainsi :

```
int on_exit (void (* fonction) (int, void *), void * argument);
```

Le second argument est souvent utilisé pour passer un pointeur de fichier `FILE *` à terminer de traiter avant de finir. Voici un exemple où la routine de terminaison ne fait que fermer le fichier transmis s'il est non `NULL`. Bien sûr, elle pourrait effectuer une tâche bien plus compliquée, comme mettre à jour un en-tête ou une table des matières en début de fichier.

exemple_on_exit.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    void gestion_sortie (int code, void * pointeur);

    int
main (void)
{
    FILE * fp;

    fp = fopen("exemple_atexit.c", "r");
    if (on_exit(gestion_sortie, (void *) fp) != 0)
        fprintf(stderr, "Erreur dans on_exit \n");

    fp = fopen("exemple_on_exit.c", "r");
    if (on_exit(gestion_sortie, (void *) fp) != 0)
        fprintf(stderr, "Erreur dans on_exit \n");

    if (on_exit(gestion_sortie, NULL) != 0)
        fprintf(stderr, "Erreur dans on_exit \n");

    fprintf(stdout, "Allez... on quitte en revenant de main()\n");
    return 4;
}
```

```
    void
gestion_sortie (int code, void * pointeur)
{
    fprintf(stdout, "Gestion Sortie appelée... code %d\n", code);
    if (pointeur == NULL) {
        fprintf(stdout, "Pas de fichier à fermer \n");
    } else {
        fprintf(stdout, "Fermeture d'un fichier \n");
        fclose((FILE *) pointeur);
    }
}
```

L'exécution suivante nous montre que les fonctions sont appelées, comme pour `atexit()`, dans l'ordre inverse de leur programmation (le pointeur `NULL` programmé en dernier apparaît en premier). Nous voyons également que le code de retour de `main()`, 4, est bien transmis à la routine de terminaison.

```
$ ./exemple_on_exit
Allez... on quitte en revenant de main()
Gestion Sortie appelée... code 4
Pas de fichier à fermer
Gestion Sortie appelée... code 4
Fermeture d'un fichier
Gestion Sortie appelée... code 4
Fermeture d'un fichier
$
```

Attendre la fin d'un processus fils

L'une des notions fondamentales dans la conception des systèmes Unix est la mise à disposition de l'utilisateur d'un grand nombre de petits utilitaires très spécialisés et très configurables grâce à des options en ligne de commande. Ces petits utilitaires peuvent être associés, par des redirections d'entrée-sortie, en commandes plus complexes, et regroupés dans des fichiers scripts simples à écrire et à déboguer.

Il est primordial dans ces scripts de pouvoir déterminer si une commande a réussi à effectuer son travail correctement ou non. On imagine donc l'importance qui peut être portée à la lecture du code de retour d'un processus. Cette importance est telle qu'un processus qui se termine passe automatiquement par un état spécial, *zombie*[1], en attendant que le processus père ait lu son code de retour. Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l'état zombie. Voici un exemple, dans lequel le processus fils attend deux secondes avant de se terminer, tandis que le processus père affiche régulièrement l'état de son fils en invoquant la commande `ps`.

exemple_zombie_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
```

1. Les différents états d'un processus seront détaillés dans le chapitre 11 consacré aux mécanismes d'ordonnancement disponibles sous Linux.

```
    int
main (void)
{
    pid_t pid;
    char  commande[128];

    if ((pid = fork()) < 0) {
        fprintf(stderr, "échec fork()\n");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        /* processus fils */
        sleep(2);
        fprintf(stdout, "Le processus fils %ld se termine \n",
                            (long) getpid());
        exit(EXIT_SUCCESS);
    } else {
        /* processus père */
        snprintf (commande, 128, "ps %ld", (long) pid);
        system(commande);
        sleep(1);
        system(commande);
        sleep(1);
        system(commande);
        sleep(1);
        system(commande);
        sleep(1);
        system(commande);
        sleep(1);
        system(commande);
    }
    return EXIT_SUCCESS;
}
```

Le "S" en deuxième colonne indique que le processus fils est endormi au début, puis il se termine et passe à l'état zombie "Z" :

```
$ ./exemple_zombie_1
  PID TTY      STAT   TIME COMMAND
  949 pts/0    S      0:00 ./exemple_zombie_1
  PID TTY      STAT   TIME COMMAND
  949 pts/0    S      0:00 ./exemple_zombie_1
Le processus fils 949 se termine
  PID TTY      STAT   TIME COMMAND
  949 pts/0    Z      0:00 [exemple_zombie_ <defunct>]
  PID TTY      STAT   TIME COMMAND
  949 pts/0    Z      0:00 [exemple_zombie_ <defunct>]
  PID TTY      STAT   TIME COMMAND
  949 pts/0    Z      0:00 [exemple_zombie_ <defunct>]
  PID TTY      STAT   TIME COMMAND
  949 pts/0    Z      0:00 [exemple_zombie_ <defunct>]
```

```
$ ps 949
  PID TTY      STAT   TIME COMMAND
$
```

Lorsque le processus père se finit, on invoque manuellement la commande `ps`, et on s'aperçoit que le fils zombie a disparu. Dans ce cas, le processus numéro 1, `init`, adopte le processus fils orphelin et lit son code de retour, ce qui provoque sa disparition. Dans ce second exemple, le processus père va se terminer au bout de 2 secondes, alors que le fils va continuer à afficher régulièrement le PID de son père.

exemple_zombie_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        fprintf (stderr, "échec fork()\n");
        exit(EXIT_FAILURE);
    }

    if (pid != 0) {
        /* processus père */
        fprintf(stdout, "Père : mon PID est %ld\n", (long)getpid());
        sleep(2);
        fprintf(stdout, "Père : je me termine \n");
        exit(EXIT_SUCCESS);
    } else {
        /* processus fils */
            fprintf(stdout, "Fils : mon père est %ld\n",
                                    (long)getppid ());
            sleep(1);
            fprintf(stdout, "Fils : mon père est %ld\n",
                                    (long)getppid ());
            sleep(1);
            fprintf(stdout, "Fils : mon père est %ld\n",
                                    (long)getppid ());
            sleep(1);
            fprintf(stdout, "Fils : mon père est %ld\n",
                                    (long)getppid ());
            sleep(1);
            fprintf(stdout, "Fils : mon père est %ld\n",
                                    (long)getppid ());
    }
    return EXIT_SUCCESS;
}
```

L'exécution suivante montre bien que le processus 1 adopte le processus fils dès que le père se termine. Au passage, on remarquera que, aussitôt le processus père terminé, le shell reprend la main et affiche immédiatement son symbole d'accueil (**$**) :

```
$ ./exemple_zombie_2
Père : mon PID est 1006
Fils : mon père est 1006
Fils : mon père est 1006
Père : je me termine
$ Fils : mon père est 1
Fils : mon père est 1
Fils : mon père est 1
```

Pour lire le code de retour d'un processus fils, il existe quatre fonctions : `wait()`, `waipid()`, `wait3()` et `wait4()`. Nous les étudierons dans cet ordre, qui est de complexité croissante. Les trois premières sont d'ailleurs des fonctions de bibliothèque implémentées en invoquant `wait4()` qui est le seul véritable appel-système.

La fonction **`wait()`** est déclarée dans `<sys/wait.h>`, ainsi :

```
pid_t wait (int * status);
```

Lorsqu'on l'invoque, elle bloque le processus appelant jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID du fils terminé. Si le pointeur `status` est non NULL, il est renseigné avec une valeur informant sur les circonstances de la mort du fils. Si un processus fils était déjà en attente à l'état zombie, `wait()` revient immédiatement. Si on n'est pas intéressé par les circonstances de la fin du processus fils, il est tout à fait possible de fournir un argument `NULL`.

La manière dont sont organisées les informations au sein de l'entier `status` est opaque, et il faut utiliser les macros suivantes pour analyser les circonstances de la fin du fils :

- **`WIFEXITED`**`(status)` est vraie si le processus s'est terminé de son propre chef en invoquant `exit()` ou en revenant de `main()`. On peut obtenir le code de retour du processus fils, c'est-à-dire la valeur transmise à `exit()`, en appelant **`WEXITSTATUS`**`(status)`.
- **`WIFSIGNALED`**`(status)` indique que le fils s'est terminé à cause d'un signal, y compris le signal `SIGABRT`, envoyé lorsqu'il appelle `abort()`. Le numéro du signal ayant tué le processus fils est disponible en utilisant la macro **`WTERMSIG`**`(status)`. À ce moment, la macro **`WCOREDUMP`**`(status)` signale si une image mémoire `core` a été créée.
- **`WIFSTOPPED`**`(status)` indique que le fils est stoppé temporairement. Le numéro du signal ayant stoppé le processus fils est accessible en utilisant **`WSTOPSIG`**`(status)`.

> **Attention**
> Les macros `WEXITSTATUS`, `WTERMSIG` et `WSTOPSIG` n'ont de sens que si les macros `WIFxxx` correspondantes ont renvoyé une valeur vraie.

La fonction `wait()` peut échouer et renvoyer –1, en plaçant l'erreur `ECHILD` dans `errno` si le processus appelant n'a pas de fils. Dans notre premier exemple, le processus père va se dédoubler en une série de fils qui se termineront de manières variées. Le processus père restera en boucle sur `wait()` jusqu'à ce qu'il ne reste plus de fils.

exemple_wait_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <sys/wait.h>

    void affichage_type_de_terminaison (pid_t pid, int status);
    int  processus_fils                (int numero_fils);

    int
main (void)
{
    pid_t pid;
    int   status;
    int   numero_fils;

    for (numero_fils = 0; numero_fils < 4; numero_fils ++) {
        switch (fork()) {
            case -1 :
                fprintf(stderr, "Erreur dans fork()\n");
                exit(EXIT_FAILURE);
            case 0 :
                fprintf(stdout, "Fils %d : PID = %ld\n",
                        numero_fils, (long)getpid());
                return processus_fils(numero_fils);
            default :
                /* processus père */
                break;
        }
    }
    /* Ici il n'y a plus que le processus père */
    while ((pid = wait(& status)) > 0)
        affichage_type_de_terminaison(pid, status);
    return EXIT_SUCCESS;
}

    void
affichage_type_de_terminaison (pid_t pid, int status)
{
    fprintf(stdout, "Le processus %ld ", (long)pid);
    if (WIFEXITED(status)) {
         fprintf(stdout, "s'est terminé normalement avec le code %d\n",
                            WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        fprintf(stdout, "s'est terminé à cause du signal %d (%s)\n",
                            WTERMSIG(status),
                            sys_siglist[WTERMSIG(status)]);
        if (WCOREDUMP(status)) {
                fprintf(stdout, "Fichier image core créé \n");
        }
    } else if (WIFSTOPPED(status)) {
```

```
        fprintf(stdout, "s'est arrêté à cause du signal %d (%s)\n",
                            WSTOPSIG(status),
                            sys_siglist[WSTOPSIG(status)]);
    }
}

    int
processus_fils (int numero_fils)
{
    switch (numero_fils) {
        case 0 :
            return 1;
        case 1 :
            exit(2);
        case 2 :
            abort();
        case 3 :
            raise(SIGUSR1);
    }
    return numero_fils;
}
```

L'exécution suivante montre bien les différents types de fin des processus fils :

```
$ ./exemple_wait_1
Fils 0 : PID = 1353
Fils 1 : PID = 1354
Fils 2 : PID = 1355
Le processus 1355 s'est terminé à cause du signal 6 (Aborted)
Fichier image core créé
Le processus 1354 s'est terminé normalement avec le code 2
Le processus 1353 s'est terminé normalement avec le code 1
Fils 3 : PID = 1356
Le processus 1356 s'est terminé à cause du signal 10 (User defined 1)
$
```

Notons qu'il n'y a pas de différence entre un retour de la fonction `main()`(fils 0, PID 1353) et un appel `exit()`(fils 1, PID 1354). De même, on voit que l'appel `abort()` (fils 2, PID 1355) se traduit bien par un envoi du signal `SIGABRT` (6 sur notre machine), avec création d'un fichier `core`. Le signal `SIGUSR1` termine le processus mais ne crée pas d'image `core`.

Il y a deux inconvénients avec la fonction `wait()`, qui ont conduit à développer la fonction `waitpid()` que nous allons voir ci-dessous. Le premier problème, c'est que l'appel reste bloquant tant qu'aucun fils ne s'est terminé. Il n'est donc pas possible d'appeler systématiquement `wait()` dans une boucle principale du programme pour savoir où en est le fils. La solution est d'installer un gestionnaire pour le signal `SIGCHLD` qui est émis dès qu'un fils se termine ou est stoppé temporairement.

Le second problème vient du fait qu'il n'est pas possible d'attendre la fin d'un fils particulier. Dans ce cas, il faut alimenter dans le gestionnaire du signal `SIGCHLD` une liste des fils terminés, qu'on consultera dans le programme principal en attente d'un processus donné. Il ne faut pas oublier de bloquer temporairement `SIGCHLD` lors de la consultation de la liste, pour éviter qu'elle ne soit modifiée pendant ce temps par l'arrivée d'un signal.

Pour pallier ces deux problèmes, un appel-système supplémentaire est disponible, **`waitpid()`**, dont le prototype est déclaré dans `<sys/wait.h>` ainsi :

```
pid_t waitpid (pid_t pid, int * status, int options);
```

Le premier argument, `pid`, permet de déterminer le processus fils dont on désire attendre la fin.

- Si `pid` est strictement positif, la fonction attend la fin du processus dont le PID correspond à cette valeur.
- Si `pid` vaut 0, on attend la fin de n'importe quel processus fils appartenant au même groupe que le processus appelant.
- Si `pid` vaut –1, on attend la fin de n'importe quel fils, comme avec la fonction `wait()`.
- Si `pid` est strictement inférieur à –1, on attend la fin de n'importe quel processus fils appartenant au groupe de processus dont le numéro est `-pid`.

Le second argument, `status`, a exactement le même rôle que `wait()`.

Le troisième argument permet de préciser le comportement de `waitpid()`, en associant par un éventuel OU binaire les constantes suivantes :

Nom	Signification
`WNOHANG`	Ne pas rester bloqué si aucun processus correspondant aux spécifications fournies par `l'argument` pid n'est terminé. Dans ce cas, `waitpid()` renverra 0.
`WUNTRACED`	Accéder également aux informations concernant les processus fils temporairement stoppés. C'est dans ce cas que les macros `WIFSTOPPED(status)` et `WSTOPSIG(status)` prennent leur signification.

Comme on le devine, il est aisé d'implémenter `wait()` à partir de `waitpid()` :

```
    pid_t
mon_wait (int * status)
{
    return waitpid(-1, status, 0);
}
```

Nous allons utiliser un programme de démonstration dans lequel un processus fils, qui reste en boucle, sera surveillé par le processus père, alors qu'un second fils, qui se termine au bout de quelques secondes, n'est pas pris en considération. Nous agirons sur une seconde console pour examiner le `status` des fils avec la commande `ps`, et pour stopper et relancer le premier fils.

exemple_wait_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <sys/wait.h>

    int
main (void)
{
```

```
    pid_t   pid;
    int     status;

    switch (pid = fork()) {
        case -1 :
            fprintf(stderr, "Erreur dans fork()\n");
            exit(EXIT_FAILURE);
        case 0 : /* fils 1 */
            fprintf(stdout, "Fils 1 : PID = %ld\n", (long)getpid());
            while (1)
                pause();
        default : /* père */
             break;
    }

    /* Créons un fils qu'on n'attend pas */
    switch (fork ()) {
        case -1 :
            fprintf(stderr, "Erreur dans fork()\n");
            exit(EXIT_FAILURE);
        case 0 : /* fils 2 */
            fprintf(stdout, "Fils 2 : PID = %ld\n", (long)getpid());
            sleep(2);
            exit(EXIT_SUCCESS);
        default : /* père */
            break;
    }
    while (1) {
        sleep(1);
        if (waitpid(pid, & status, WUNTRACED | WNOHANG) > 0) {
            if (WIFEXITED(status)) {
                fprintf(stdout, "%ld terminé par exit (%d)\n",
                               (long)pid, WEXITSTATUS(status));
                exit(EXIT_SUCCESS);
            } else if (WIFSIGNALED (status)) {
                fprintf(stdout, "%ld terminé par signal %d\n",
                               (long)pid, WTERMSIG(status));
                exit(EXIT_SUCCESS);
            } else if (WIFSTOPPED(status)) {
                fprintf(stdout, "%ld stoppé par signal %d\n",
                               (long)pid, WSTOPSIG(status));
            }
        }
    }
    return EXIT_SUCCESS;
}
```

L'exécution suivante montre en seconde colonne les actions depuis l'autre Xterm :

```
$ ./exemple_wait_2
Fils 1 : PID = 1525
Fils 2 : PID = 1526
```

```
                    $ ps 1525 1526
                     PID TTY      STAT COMMAND
                    1525 pts/0    S    ./exemple_wait_2
                    1526 pts/0    Z   [exemple_wait_2 <defunct>]
                    $ kill -STOP 1525
1525 stoppé par signal 19
                    $ ps 1525 1526
                     PID TTY      STAT COMMAND
                    1525 pts/0    T    ./exemple_wait_2
                    1526 pts/0    Z    [exemple_wait_2 <defunct>]
                    $ kill -CONT 1525
                    $ kill -TSTP 1525
1525 stoppé par signal 20
                    $ kill -CONT 1525
                    $ kill -TERM 1525
1525 terminé par signal 15
$
```

Nous voyons que le fils 2, de PID 1526, reste à l'état zombie dès qu'il se finit car le père ne demande pas son code de retour. Le fait d'avoir appelé l'option `WUNTRACED` nous permet d'être informé lorsque le processus fils 1 est temporairement stoppé par un signal.

Les fonctions `wait()` et `waitpid()` sont définies par SUSv3, contrairement aux deux autres fonctions que nous allons étudier, `wait3()` et `wait4()`, qui sont d'inspiration BSD. Elles permettent, par rapport à `wait()` ou `waitpid()`, d'obtenir des informations supplémentaires sur le processus qui s'est terminé. Ces renseignements sont transmis par l'intermédiaire d'une structure `rusage`, définie dans `<sys/resource.h>`.

Cette structure regroupe des statistiques sur l'utilisation par le processus des ressources système. Le type `struct rusage` contient les champs suivants :

Type	Nom	Signification
`struct timeval`	`ru_utime`	Temps passé par le processus en mode utilisateur.
`struct timeval`	`ru_stime`	Temps passé par le processus en mode noyau.
`long`	`ru_maxrss`	Taille maximale des données placées simultanément en mémoire (exprimée en Ko).
`long`	`ru_ixrss`	Taille de la mémoire partagée avec d'autres processus (exprimée en Ko).
`long`	`ru_idrss`	Taille des données non partagées (en Ko).
`long`	`ru_isrss`	Taille de la pile exprimée en Ko.
`long`	`ru_minflt`	Nombre de fautes de pages mineures (n'ayant pas nécessité de rechargement depuis le disque).
`long`	`ru_majflt`	Nombre de fautes de pages majeures (ayant nécessité un rechargement des données depuis le disque).
`long`	`ru_nswap`	Nombre de fois où le processus a été entièrement swappé.
`long`	`ru_inblock`	Nombre de fois où des données ont été lues.
`long`	`ru_oublock`	Nombre de fois où des données ont été écrites.
`long`	`ru_msgsnd`	Nombre de messages envoyés par le processus.
`long`	`ru_msgrcv`	Nombre de messages reçus par le processus.

Type	Nom	Signification
long	ru_nsignals	Nombre de signaux reçus par le processus
long	ru_nvcsw	Nombre de fois où le programme a volontairement cédé le processeur en attendant la disponibilité d'une ressource.
long	ru_nivcsw	Nombre de fois où le processus a été interrompu par l'ordonnanceur car il était arrivé à la fin de sa tranche de temps impartie, ou qu'un processus plus prioritaire était prêt.

La structure rusage est assez riche, malheureusement Linux ne remplit que très peu de champs. Les seules informations renvoyées sont les suivantes :

- Les temps ru_utime et ru_stime.
- Les nombres de fautes de pages ru_minflt et ru_majflt.
- Le nombre de fois où le processus a été swappé ru_nswap.

Les données fournies par cette structure sont surtout utiles lors de la mise au point d'applications assez critiques, si on désire suivre très précisément la gestion mémoire ou les entrées-sorties d'un processus.

La structure timeval qu'on rencontre dans les deux premiers champs se décompose elle-même ainsi (<sys/time.h>) :

Type	Nom	Signification
long	tv_sec	Nombre de secondes
long	tv_usec	Nombre de microsecondes

Suivant les bibliothèques utilisées, les membres de la structure timeval ont parfois un autre type que long int, notamment time_t. Quoi qu'il en soit, ces types de données peuvent être affichées comme des entiers longs, avec le format « %ld » de fprintf().

Le prototype de la fonction **wait3()** est le suivant :

```
pid_t wait3 (int * status, int options, struct rusage * rusage);
```

Les options de wait3() sont les mêmes que celles de waitpid(), c'est-à-dire WNOHANG et WUNTRACED. Le status est également le même qu'avec wait() ou waitpid(), et on utilise les mêmes macros pour l'analyser.

Nous pourrions définir wait() en utilisant wait3(status, 0, NULL).

Dans notre exemple, nous allons afficher les valeurs de la structure rusage représentant le temps passé par le processus en mode utilisateur et le temps passé en mode noyau.

exemple_wait_3.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/resource.h>
```

```
    int
main (void)
{
    pid_t   pid;
    int     status;
    struct rusage usage;
    int     i, j;

    switch (pid = fork()) {
        case -1 :
            fprintf(stderr, "Erreur dans fork()\n");
            exit(EXIT_FAILURE);
        case 0 : /* fils */
            fprintf(stdout, "Fils : PID = %ld\n", (long)getpid());
            j = 0;
            for (i = 0; i < 5000000; i ++)
                j += i;
            raise(SIGSTOP);
            for (i = 0; i < 500000; i ++) {
                FILE * fp;
                fp = fopen("exemple_wait_2", "r");
                if (fp != NULL)
                    fclose(fp);
            }
            exit(EXIT_SUCCESS);
        default : /* père */
            break;
    }

    while (1) {
        sleep(1);
        if ((pid = wait3(& status, WUNTRACED | WNOHANG, & usage)) > 0) {
            if (WIFEXITED(status)) {
                 fprintf(stdout, "%ld terminé par exit (%d)\n",
                                  (long)pid, WEXITSTATUS(status));
            } else if (WIFSIGNALED (status)) {
                 fprintf(stdout, "%ld terminé par signal %d\n",
                                  (long)pid, WTERMSIG(status));
            } else if (WIFSTOPPED (status)) {
                 fprintf(stdout, "%ld stoppé par signal %d\n",
                                  (long)pid, WSTOPSIG(status));
                 fprintf(stdout, "Je le relance \n");
                 kill(pid, SIGCONT);
            }
            fprintf(stdout, "Temps utilisateur %ld s, %ld µs\n",
                           usage.ru_utime.tv_sec,
                           usage.ru_utime.tv_usec);
            fprintf(stdout, "Temps en mode noyau %ld s, %ld µs\n",
                           usage.ru_stime.tv_sec,
                           usage.ru_stime.tv_usec);
        } else if (errno == ECHILD) {
              /* Plus de fils */
```

```
                break;
            }
        }
        return EXIT_SUCCESS;
    }
```

Le processus fils effectue une boucle, en mode utilisateur uniquement, avant de se stopper lui-même. Le processus père le relance alors. Le fils passe ensuite à une boucle au sein de laquelle il fait des appels-système, et fonctionne donc en mode noyau. Voici un exemple d'exécution :

```
$ ./exemple_wait_3
Fils : PID = 2017
2017 stoppé par signal 19
Je le relance
Temps utilisateur 0 s, 100000 µs
Temps en mode noyau 0 s, 0 µs
2017 terminé par exit (0)
Temps utilisateur 1 s, 610000 µs
Temps en mode noyau 2 s, 290000 µs
$
```

La dernière fonction de cette famille est **`wait4()`**. Elle permet d'obtenir à la fois des statistiques, comme avec `wait3()`, et d'attendre un processus fils particulier, comme `waitpid()`. Le prototype de cette fonction est le suivant :

```
pid_t wait4 (pid_t pid, int * status,
             int options, struct rusage * rusage);
```

La fonction `wait4()` est la seule qui soit un véritable appel-système sous Linux ; les autres fonctions de cette famille en découlent ainsi :

Fonction	Implémentation
`wait3 (status, options, usage)`	`wait4 (-1, status, options, usage)`
`waitpid (pid, status, options)`	`wait4 (pid, status, options, NULL)`
`wait (status)`	`wait4 (-1, status, 0, NULL)`

Depuis Linux 2.6, un nouvel appel-système a fait son apparition : **`waitid()`**. Il permet en outre d'attendre qu'un processus fils redémarre après un arrêt, et d'obtenir les informations de `status` dans un format différent (une structure `siginfo_t` que nous verrons dans le chapitre 8). Les constantes symboliques nécessaires au fonctionnement de `waitid` n'étant pas disponibles dans la bibliothèque C au moment de la rédaction de ces lignes, nous ne le détaillerons pas plus. Le lecteur intéressé pourra se reporter au standard SUSv3 qui décrit son fonctionnement.

Signaler une erreur

Il y a des cas où la gestion d'erreur doit être moins drastique qu'un arrêt anormal du programme. Pour cela, les appels-système remplissent la variable globale externe **`errno`** avec une valeur numérique entière représentant le type d'erreur qui s'est produite. Toutes les valeurs représentant une erreur sont non nulles.

Attention

Le fait que `errno` soit remplie avec une valeur non nulle n'est pas suffisant pour en déduire qu'une erreur s'est produite. Il faut pour cela que l'appel-système échoue explicitement (la plupart du temps en renvoyant –1 et non pas 0). Ceci est encore plus vrai avec des routines de bibliothèque qui peuvent invoquer plusieurs appels-système et remédier aux conditions d'erreur. `errno` sera alors modifiée à plusieurs reprises avant le retour de la fonction. SUSv3 autorise les fonctions de bibliothèque à modifier `errno` même lorsqu'elles réussissent à faire leur travail.

Il existe des constantes symboliques représentant chaque erreur possible. Elles sont définies dans le fichier `<errno.h>`. Toutefois, celui-ci inclut automatiquement `<bits/errno.h>`, `<linux/errno.h>` et `<asm/errno.h>`, qui définissent l'essentiel des constantes d'erreur. Il est bon de connaître l'existence de ces fichiers car un coup d'œil rapide permet parfois d'identifier une erreur qu'on n'avait pas prévue, à partir de son numéro. Les principales erreurs qu'on rencontre fréquemment dans les appels-système sont décrites dans le tableau ci-dessous. Nous en avons limité la liste aux plus courantes. Il en existe de nombreuses autres, par exemple dans le domaine de la programmation réseau, que nous détaillerons le moment venu.

Nom	Signification
`E2BIG`	La liste d'arguments fournie à l'une des fonctions de la famille `exec()` est trop longue.
`EACCES`	L'accès demandé est interdit, par exemple dans une tentative d'ouverture de fichier avec `open()`.
`EAGAIN`	L'opération est momentanément impossible, il faut réessayer. Par exemple, on demande une lecture non bloquante avec `read()`, mais aucune donnée n'est encore disponible.
`EBADF`	Le descripteur de fichier transmis à l'appel-système, par exemple `close()`, est invalide.
`EBUSY`	Le répertoire ou le fichier considéré est en cours d'utilisation. Ainsi, `umount()` ne peut démonter un périphérique si un processus l'utilise alors comme répertoire de travail.
`ECHILD`	Le processus attendu par `waitpid()` ou `wait4()` n'existe pas ou n'est pas un fils du processus appelant.
`EDEADLK`	Le verrouillage en écriture par `fcntl()` du fichier demandé conduirait à un blocage.
`EDOM`	La valeur transmise à la fonction mathématique est hors de son domaine de définition. Par exemple, on appelle `acos()` avec une valeur inférieure à -1 ou supérieure à 1.
`EEXIST`	Le fichier ou le répertoire indiqué pour une création existe déjà. Par exemple, avec `open()`, `mkdir()`, `mknod()`…
`EFAULT`	Un pointeur transmis en argument pointe en dehors de l'espace d'adressage du processus. Cette erreur révèle un bogue grave dans le programme.
`EFBIG`	On a tenté de créer un fichier de taille plus grande que la limite autorisée pour le processus.
`EINTR`	L'appel-système a été interrompu par l'arrivée d'un signal qui a été intercepté par un gestionnaire installé par le processus.
`EINVAL`	Un argument de type entier, ou représenté par une constante symbolique, a une valeur invalide ou incohérente.
`EIO`	Une erreur d'entrée-sortie de bas niveau s'est produite pendant un accès au fichier.
`EISDIR`	Le descripteur de fichier transmis à l'appel-système, par exemple `read()`, correspond à un répertoire.
`ELOOP`	On a rencontré trop de liens symboliques successifs, il y a probablement une boucle entre eux.

Nom	Signification
EMFILE	Le processus a atteint le nombre maximal de fichiers ouverts simultanément.
EMLINK	On a déjà créé le nombre maximal de liens sur un fichier.
ENAMETOOLONG	Le chemin d'accès transmis en argument, par exemple pour `chdir()`, est trop long.
ENFILE	On a atteint le nombre maximal de fichiers ouverts simultanément sur l'ensemble du système.
ENODEV	Le fichier spécial de périphérique n'est pas valide, par exemple dans l'appel-système `mount()`.
ENOENT	Un répertoire contenu dans le chemin d'accès fourni à l'appel-système n'existe pas, ou est un lien symbolique pointant dans le vide.
ENOEXEC	Le fichier exécutable indiqué à l'un des appels de la famille `exec()` n'est pas dans un format reconnu par le noyau.
ENOLCK	Il n'y a plus de place dans la table système pour ajouter un verrou avec l'appel-système `fcntl()`.
ENOMEM	Il n'y a plus assez de place mémoire pour allouer une structure supplémentaire dans une table système.
ENOSPC	Le périphérique sur lequel on veut créer un nouveau fichier ou écrire des données supplémentaires n'a plus de place disponible.
ENOSYS	La fonctionnalité demandée par l'appel-système n'est pas disponible dans le noyau. Il peut s'agir d'un problème de version ou d'options lors de la compilation du noyau.
ENOTBLK	Le fichier spécial qu'on tente de monter avec `mount()` ne représente pas un périphérique de type « bloc ». Cette erreur n'est pas décrite dans SUSv3.
ENOTDIR	Un élément du chemin d'accès fourni n'est pas un répertoire.
ENOTEMPTY	Le répertoire qu'on veut détruire n'est pas vide, ou le nouveau nom d'un répertoire à renommer existe déjà et n'est pas vide.
ENOTTY	Le fichier indiqué en argument à `ioctl()` n'est pas un terminal.
ENXIO	Le fichier spécial indiqué n'est pas reconnu par le noyau (par exemple un numéro de nœud majeur invalide).
EPERM	Le processus appelant n'a pas les autorisations nécessaires pour effectuer l'opération prévue. Souvent, il s'agit d'une fonctionnalité réservée à *root*.
EPIPE	Tentative d'écriture avec `write()` dans un tube dont l'autre extrémité a été fermée par le processus lecteur. Cette erreur n'est envoyée que si le signal `SIGPIPE` est bloqué ou ignoré.
ERANGE	Une valeur numérique attendue dans une fonction mathématique est invalide.
EROFS	On tente une modification sur un fichier appartenant à un système de fichiers monté en lecture seule.
ESPIPE	On essaye de déplacer le pointeur de lecture, avec `lseek()`, sur un descripteur de fichier ne le permettant pas, comme un tube ou une socket.
ESRCH	Le processus visé, par exemple avec `kill()`, n'existe pas.
ETXTBSY	On essaye d'exécuter un fichier déjà ouvert en écriture par un autre processus.
EWOULDBLOCK	Synonyme de `EAGAIN` qu'on rencontre dans la description de nombreuses fonctionnalités réseau.
EXDEV	On essaye de renommer un fichier ou de créer un lien matériel entre deux systèmes de fichier différents.

On voit, à l'énoncé d'une telle liste (qui ne représente qu'un tiers environ de toutes les erreurs pouvant se produire sous Linux), qu'il est difficile de gérer tous les cas à chaque appel-système effectué par le programme.

Alors que faire si, par exemple, l'appel-système `open()` échoue ? À lui seul, il peut déjà renvoyer une bonne quinzaine d'erreurs différentes. Tout dépend du degré de convivialité du logiciel développé. Dans certains cas, on peut se contenter de mettre un message sur la sortie d'erreur « impossible d'ouvrir le fichier xxx », et arrêter le programme. À l'opposé, on peut aussi diagnostiquer qu'un des répertoires du chemin d'accès est invalide et afficher pour l'utilisateur la liste des répertoires du même niveau pour qu'il corrige son erreur.

Ainsi, certaines erreurs ne doivent jamais se produire dans un programme bien débogué. C'est le cas de `EFAULT`, qui signale un pointeur mal initialisé, de `EDOM` ou `ERANGE`, qui indiquent qu'une fonction mathématique a été appelée sans vérifier si les variables appartiennent à son domaine de définition. Ces cas-là peuvent être contrôlés dans des appels à `assert()`, car il s'agit de bogues à éliminer avant la distribution du logiciel.

Dans d'autres cas, le problème concerne le système, et le pauvre programme ne peut rien faire pour corriger l'erreur. C'est le cas par exemple `de ENOMEM`, qui indique que le noyau n'a plus assez de place mémoire, ou `de ENFILE`, qui se produit lorsque le nombre maximal de fichiers ouverts sur le système est atteint. Il n'y a guère d'autres possibilités alors que d'abandonner l'opération après avoir signalé le problème à l'utilisateur.

La règle absolue est de ne jamais passer sous silence les conditions d'erreur qui paraissent improbables. Si une application doit être distribuée largement et utilisée pendant de longues heures par ses utilisateurs, il est pratiquement certain qu'elle sera un jour confrontée au problème d'une partition disque saturée. Ignorer le code de retour de `write()` reviendra à ne pas sauvegarder le travail de l'utilisateur, alors qu'un simple message d'avertissement lui aurait permis d'effacer des fichiers inutiles et de refaire une sauvegarde.

Si on ne désire pas traiter au cas par cas toutes les erreurs possibles, on peut employer la fonction **`strerror()`**, déclarée dans `<string.h>` ainsi :

```
char * strerror (int numero_erreur);
```

Cette fonction renvoie un pointeur sur une chaîne de caractères statique décrivant l'erreur produite. Cette chaîne de caractères peut être écrasée à chaque nouvel appel de `strerror()`. Pour éviter ce problème dans le cas d'une programmation multi-thread, on peut utiliser l'extension Gnu **`strerror_r()`**, déclarée ainsi :

```
char * strerror_r (int numero_erreur, char * chaine, size_t longueur)
```

Cette fonction n'écrit jamais dans la chaîne plus d'octets que la longueur indiquée, y compris le caractère nul final.

Dans tous les cas, il convient de consulter la page de manuel des appels-système et des fonctions de bibliothèque employés (en espérant que les informations soient à jour), et de prévoir une gestion adéquate pour les erreurs les plus fréquentes. Une gestion générique peut être mise en place pour les cas les plus rares. Prenons l'exemple de `open()`. Une manière assez simple mais correcte d'opérer serait :

```
    while (1) {
        if ((fd = open(fichier, mode)) == -1) {
            assert(errno != EFAULT);
            switch (errno) {
                case EMFILE :
                case ENFILE :
                case ENOMEM :
```

```
                    fprintf(stderr, "Erreur critique : %s\n",
                              strerror(errno));
                    return -1;
                default :
                    fprintf(stderr, "Erreur d'ouverture de %s : %s\n",
                              fichier, strerror(errno));
                    break;
            }
            if (corriger_le_nom_pour_reessayer() != 0)
                /* l'utilisateur préfère abandonner */
                return -1;
            else
                continue; /* recommencer */
        } else {
            /* pas d'erreur */
            break;
    }
    return 0;
}
```

Cela permet à la fois de différencier les erreurs irrécupérables de celles qu'on peut corriger, et donne à l'utilisateur la possibilité de modifier sa demande ou d'abandonner l'opération.

Nous allons voir un petit exemple d'utilisation de `strerror()`, en l'invoquant pour une dizaine d'erreurs courantes :

exemple_strerror.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <errno.h>

    int
main (void)
{
    fprintf(stdout, "strerror(EACCES) = %s\n", strerror(EACCES));
    fprintf(stdout, "strerror(EAGAIN) = %s\n", strerror(EAGAIN));
    fprintf(stdout, "strerror(EBUSY)  = %s\n", strerror(EBUSY) );
    fprintf(stdout, "strerror(ECHILD) = %s\n", strerror(ECHILD));
    fprintf(stdout, "strerror(EEXIST) = %s\n", strerror(EEXIST));
    fprintf(stdout, "strerror(EFAULT) = %s\n", strerror(EFAULT));
    fprintf(stdout, "strerror(EINTR)  = %s\n", strerror(EINTR) );
    fprintf(stdout, "strerror(EINVAL) = %s\n", strerror(EINVAL));
    fprintf(stdout, "strerror(EISDIR) = %s\n", strerror(EISDIR));
    fprintf(stdout, "strerror(EMFILE) = %s\n", strerror(EMFILE));
    fprintf(stdout, "strerror(ENODEV) = %s\n", strerror(ENODEV));
    fprintf(stdout, "strerror(ENOMEM) = %s\n", strerror(ENOMEM));
    fprintf(stdout, "strerror(ENOSPC) = %s\n", strerror(ENOSPC));
    fprintf(stdout, "strerror(EPERM)  = %s\n", strerror(EPERM) );
    fprintf(stdout, "strerror(EPIPE)  = %s\n", strerror(EPIPE) );
    fprintf(stdout, "strerror(ESRCH)  = %s\n", strerror(ESRCH) );
```

```
    return EXIT_SUCCESS;
}
```

L'exécution montre que la fonction `strerror()` de la GlibC est sensible à la localisation :

```
$ echo $LC_ALL
fr_FR
$ ./exemple_strerror
strerror(EACCES) = Permission non accordée
strerror(EAGAIN) = Ressource temporairement non disponible
strerror(EBUSY)  = Périphérique ou ressource occupé
strerror(ECHILD) = Aucun processus enfant
strerror(EEXIST) = Le fichier existe
strerror(EFAULT) = Mauvaise adresse
strerror(EINTR)  = Appel-système interrompu
strerror(EINVAL) = Paramètre invalide
strerror(EISDIR) = Est un répertoire
strerror(EMFILE) = Trop de fichiers ouverts
strerror(ENODEV) = Aucun périphérique de ce type
strerror(ENOMEM) = Ne peut allouer de la mémoire
strerror(ENOSPC) = Aucun espace disponible sur le périphérique
strerror(EPERM)  = Opération non permise
strerror(EPIPE)  = Relais brisé (pipe)
strerror(ESRCH)  = Aucun processus de ce type
$
```

Il existe également une fonction permettant d'afficher directement sur la sortie standard, précédée éventuellement d'une chaîne de caractères permettant de situer l'erreur. Cette fonction, nommée **`perror()`**, est déclarée dans `<stdio.h>` ainsi :

```
void perror (const char * s);
```

On l'utilise généralement de la manière suivante :

```
    if ((fd = open(nom_fichier, mode)) == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
```

Le message s'affiche ainsi :

```
open: Aucun fichier ou répertoire de ce type
```

Il s'agit dans ce cas de l'erreur `ENOENT`.

Notre exemple va utiliser `perror()` en cas d'échec de `fork()`. Pour faire échouer celui-ci, nous allons diminuer la limite `RLIMIT_NPROC`[1], puis nous bouclerons sur un appel `fork()`.

exemple_perror.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/resource.h>
```

1. Les limites d'exécution des processus sont étudiées dans le chapitre 9.

```
    #include <sys/wait.h>

    int
main (void)
{
    struct rlimit limite;
    pid_t   pid;

    if (getrlimit(RLIMIT_NPROC, & limite) != 0) {
        perror("getrlimit");
        exit(EXIT_FAILURE);
    }
    limite.rlim_cur = 16;
    if (setrlimit(RLIMIT_NPROC, & limite) != 0) {
        perror("setrlimit");
        exit(EXIT_FAILURE);
    }
    while (1) {
        pid = fork();
        if (pid == (pid_t) -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (pid != 0) {
            fprintf(stdout, "%ld\n", (long)pid);
            if (waitpid(pid, NULL, 0) != pid)
                perror("waitpid");
            break;
        }
    }
    return EXIT_SUCCESS;
}
```

Comme nous avons déjà plusieurs processus qui tournent avant même de lancer le programme, nous n'atteindrons pas les seize `fork()`. Pour vérifier le nombre de processus en cours, nous allons d'abord lancer une série de commandes shell, avant d'exécuter le programme.

```
$ ps aux | grep `whoami` |wc -l
     10
$ ./exemple_perror
6922
6923
6924
6925
6926
6927
6928
fork: Ressource temporairement non disponible
$
```

Dans les dix processus qui tournaient avant le lancement, il faut compter la commande `ps` elle-même. Il est donc normal que `fork()` n'échoue qu'une fois arrivé au septième processus fils.

Le message correspond à celui de l'erreur `EAGAIN`. En effet, le système considère que l'un des processus va se finir tôt ou tard et que `fork()` pourra réussir alors.

Un autre moyen d'accéder directement aux messages d'erreur est d'employer la table **`sys_errlist[]`**, définie dans `<errno.h>` ainsi :

```
const char * sys_errlist [];
```

Chaque élément de la table est un pointeur sur une chaîne de caractères décrivant l'erreur correspondant à l'index de la chaîne dans la table. Il existe une variable globale décrivant le nombre d'entrées dans la table :

```
int sys_nerr;
```

Il faut être très prudent avec les accès dans cette table car il y a des valeurs ne correspondant à aucune erreur. C'est le cas, par exemple, de 41 et 58 sous Linux avec la GlibC 2. Pareillement, dans la même configuration, `sys_nerr` vaut 125, ce qui signifie que les erreurs s'étendent de 0 à 124. Pourtant, il existe une erreur `ECANCELED` valant 125, non utilisée dans les appels-système.

L'accès à la table doit donc être précautionneux, du genre :

```
    if ((erreur < 0) || (erreur >= sys_nerr)) {
        fprintf(stderr, "Erreur invalide %d\n", erreur);
    } else if (sys_errlist[erreur] == NULL) {
        fprintf(stderr, "Erreur non documentée %d\n", erreur);
    } else {
        fprintf(stderr, "%s\n", sys_errlist[erreur]);
    }
```

Nous allons utiliser la table `sys_errlist[]` pour afficher tous les libellés des erreurs connues.

exemple_sys_errlist.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <errno.h>

    int
main (void)
{
    int i;
    for (i = 0; i < sys_nerr; i++)
        if (sys_errlist[i] != NULL)
            fprintf(stdout, "%d : %s\n", i, sys_errlist[i]);
        else
            fprintf(stdout, "** Pas de message pour %d **\n", i);
    return EXIT_SUCCESS;
}
```

Dans l'exemple d'exécution suivant, nous avons éliminé quelques passages pour éviter d'afficher inutilement toute la liste :

```
$ ./exemple_sys_errlist
0 : Success
1 : Operation not permitted
```

```
2 : No such file or directory
3 : No such process
4 : Interrupted system call
5 : Input/output error
6 : Device not configured
7 : Argument list too long
8 : Exec format error
9 : Bad file descriptor
10 : No child processes
  [...]
39 : Directory not empty
40 : Too many levels of symbolic links
** Pas de message pour 41 **
42 : No message of desired type
  [...]
56 : Invalid request code
57 : Invalid slot
** Pas de message pour 58 **
59 : Bad font file format
60 : Device not a stream
61 : No data available
62 : Timer expired
[...]
121 : Remote I/O error
122 : Disk quota exceeded
123 : No medium found
124 : Wrong medium type
$
```

On remarque plusieurs choses : d'abord les messages de `sys_errlist[]` ne sont pas traduits automatiquement par la localisation, contrairement à `strerror()`, ensuite l'erreur 0, bien que documentée pour simplifier l'accès à cette table, n'est par définition pas une erreur, enfin les erreurs 41 et 58 n'existent effectivement pas.

Conclusion

Nous avons étudié dans ce chapitre les principaux points importants concernant la fin d'un processus, due à des causes normales ou à un problème irrémédiable.

Nous avons également essayé de définir un comportement raisonnable en cas de détection d'erreur, et nous avons analysé les méthodes pour obtenir des informations sur les raisons ayant conduit un processus fils à se terminer. En ce qui concerne le code de débogage, et les macros `assert()`, on trouvera des réflexions intéressantes dans [McConnell 1994] *Programmation professionnelle*.

Nous avons aussi indiqué qu'un processus pouvait être tué par un signal. Nous allons à présent développer ce sujet dans les quelques chapitres à venir.

6

Gestion classique des signaux

La gestion des signaux entre processus est peut-être la partie la plus passionnante de la programmation sous Unix. C'est aussi celle qui peut conduire aux dysfonctionnements les plus subtils, avec des bogues très difficiles à détecter de par leur nature fondamentalement intempestive.

On peut traiter les signaux de deux façons : une classique, en partie définie par la norme Ansi C, que nous étudierons dans ce chapitre, et une plus performante, définie à l'origine par les normes Posix.1 et Posix.1b puis de nos jours par SUSv3, ainsi que nous le verrons dans les prochains chapitres.

Généralités

Le principe est *a priori* simple : un processus peut envoyer sous certaines conditions un signal à un autre processus (ou à lui-même). Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement (aux délais dus à l'ordonnancement près) une mesure spécifique. Le destinataire peut soit ignorer le signal, soit le capturer – c'est-à-dire dérouter provisoirement son exécution vers une routine particulière qu'on nomme gestionnaire de signaux –, soit laisser le système traiter le signal avec un comportement par défaut.

La plupart des signaux qui nous intéressent ne sont pas émis par des processus applicatifs, mais directement par le noyau en réponse à des conditions logicielles ou matérielles particulières. Certains signaux font partie d'une classe nouvelle, les signaux temps-réel, définis par la norme Posix.1b. Nous les étudierons à part, dans le chapitre 8.

Il existe un nombre déterminé de signaux (32 sous Linux 2.0, 64 depuis Linux 2.2). Chaque signal dispose d'un nom défini sous forme de constante symbolique commençant par `SIG` et d'un numéro associé. Il n'y a pas de nom pour le signal numéro 0, car cette valeur a un rôle particulier que nous verrons plus tard. Toutes les définitions concernant les signaux se trouvent dans le fichier d'en-tête `<signal.h>` (ou dans d'autres fichiers qu'il inclut lui-même).

Il est important de toujours utiliser le nom symbolique du signal et non son numéro, car celui-ci peut varier d'un système à l'autre, voire selon la machine employée, même avec une version identique du noyau. La constante symbolique **NSIG** définie par la bibliothèque C correspond au nombre de signaux, y compris un pseudo-signal numéroté 0. Il arrive que, sur certains systèmes, cette constante soit nommée **_NSIG**. Pour assurer la portabilité d'un programme, on peut donc inclure en début de fichier des directives pour le préprocesseur, du type :

```
#include <signal.h>

    #ifndef NSIG
    #ifndef _NSIG
        #error "NSIG et _NSIG indéfinis"
    #else
        #define NSIG _NSIG
    #endif
#endif
```

Sur les systèmes supportant les signaux temps-réel, comme Linux, il existe deux valeurs supplémentaires importantes : **SIGRTMIN** et **SIGRTMAX**. Il s'agit des numéros du plus petit et du plus grand signal temps-réel. Ces derniers en effet s'étendent sur une plage continue en dessous de NSIG-1. Sous Linux, l'organisation des signaux est la suivante :

Valeur	Signal
0 ... 31	Signaux classiques (non temps-réel)
32 ... 63	SIGRTMIN (signaux temps-réel) SIGRTMAX
64	NSIG

Attention

SIGRTMIN et SIGRTMAX ne sont pas nécessairement des constantes symboliques du préprocesseur. SUSv3 autorise leur implémentation sous forme de variables. Leur valeur n'est donc pas toujours disponible à la compilation. Par contre, la constante symbolique _POSIX_REALTIME_SIGNALS doit être définie dans <unistd.h>.

Il nous arrivera de vouloir balayer uniquement la liste des signaux classiques. Pour cela, nous définirons une variable locale NB_SIG_CLASSIQUES permettant de travailler sur tous les systèmes :

```
#ifdef _POSIX_REALTIME_SIGNALS
    #define NB_SIG_CLASSIQUES  SIGRTMIN
#else
    #define NB_SIG_CLASSIQUES  NSIG
#endif
```

Chaque signal classique a une signification bien précise, et les processus ont par défaut un comportement adapté à la situation représentée par le signal. Par exemple, le signal indiquant

la fin d'un processus fils est ignoré par défaut. Par contre, la déconnexion du terminal de contrôle envoie un signal qui arrête le processus, et une référence mémoire invalide termine le processus et crée un fichier d'image mémoire (`core`) permettant le débogage du programme.

Pour qu'un fichier `core` soit créé, un certain nombre de conditions sont nécessaires : permission d'écrire dans le répertoire de lancement du processus, pas d'exécution *Set-UID* ou *Set-GID* et pas de limitation dans la taille de ces fichiers. Cette limite est fixée avant le lancement du processus par la commande shell `ulimit`. Ainsi, il faudra penser à exécuter la commande :

```
$ ulimit -c unlimited
```

avant de démarrer une session de débogage, pour s'assurer que les fichiers `core` soient créés sans limite de taille. (Cette opération peut également être réalisée directement dans le programme, comme nous le verrons dans le chapitre 9.)

Avant d'étudier les mécanismes d'émission et de réception des signaux, nous allons analyser précisément ceux qui sont disponibles sous Linux, en observant les conditions dans lesquelles les signaux sont émis, le comportement par défaut d'un processus qui les reçoit, et l'intérêt éventuel d'installer un gestionnaire pour les capturer.

Liste des signaux sous Linux

Certains signaux sont révélateurs de conditions d'erreur. Ils sont en général délivrés immédiatement après la détection du dysfonctionnement. On ne peut donc pas vraiment parler de comportement asynchrone. Par contre, tous les signaux indiquant une interaction avec l'environnement (action sur le terminal, connexion réseau…) ont une nature fortement asynchrone. Ils peuvent se produire à tout moment du programme.

Signaux SIGABRT et SIGIOT

Il s'agit de deux synonymes sous Linux. `SIGIOT` est le nom historique sous Système V, mais `SIGABRT` est préférable car il est défini par les normes Ansi C et SUSv3. `SIGABRT` est déclenché lorsqu'on appelle la fonction `abort()` de la bibliothèque C. Le comportement par défaut est de terminer le programme en créant un fichier `core`. La routine `abort()` sert, comme nous l'avons vu dans le chapitre précédent, à indiquer la fin anormale d'un programme (détection d'un bogue interne, par exemple).

Il est possible d'ignorer le signal `SIGABRT` (si on le reçoit depuis un autre processus), mais il faut savoir que la fonction `abort()` restitue le comportement par défaut avant d'envoyer `SIGABRT` vers le processus appelant lui-même.

La routine `abort()` de la bibliothèque C est complexe, car la norme SUSv3 réclame plusieurs fonctionnalités assez difficiles à concilier :

- Si le processus ignore `SIGABRT`, `abort()` doit réinitialiser le comportement par défaut avant d'envoyer ce signal.
- Si le processus capture le signal `SIGABRT` et si son gestionnaire redonne ensuite la main à la routine `abort()`, celle-ci doit vider tous les flux d'entrée-sortie en utilisant `fflush()` et les fermer avant de terminer le programme.
- Si le processus capture le signal `SIGABRT` et s'il ne rend pas la main à `abort()`, celle-ci ne doit pas toucher aux flux d'entrée-sortie. Pour ne pas revenir, le gestionnaire peut soit sortir

directement du programme en appelant `exit()` ou `_exit()`, soit effectuer un saut non local `siglongjmp()` vers une autre routine.

Rajoutons à tout ceci que la routine `abort()` de la bibliothèque C doit traiter le cas des programmeurs distraits qui invoquent `abort()` depuis le gestionnaire de signaux `SIGABRT` lui-même, ainsi que l'appel simultané depuis plusieurs threads. La complexité de cette tâche est toutefois résolue par la routine `abort()`, dont on peut examiner l'implémentation dans les sources de la bibliothèque GlibC.

Nous parlerons souvent des sauts non locaux au cours de ce chapitre. Nous les détaillerons lorsque nous étudierons la manière de terminer un gestionnaire de signaux. Pour le moment, on peut considérer qu'il s'agit d'une sorte de « `goto` » permettant de sauter d'une fonction au sein d'une autre, en nettoyant également la pile.

Signaux SIGALRM, SIGVTALRM et SIGPROF

`SIGALRM` (attention à l'orthographe…) est un signal engendré par le noyau à l'expiration du délai programmé grâce à l'appel-système `alarm()`. Nous l'étudierons plus en détail ultérieurement car il est souvent utilisé pour programmer une limite de temporisation pour des appels-système bloquants (comme une lecture depuis une connexion réseau). On programme un délai maximal avant d'invoquer l'appel-système susceptible de rester coincé, et l'arrivée du signal `SIGALRM` l'interrompt, avec un code d'erreur `EINTR` dans `errno`.

`SIGALRM` est également utilisé pour la programmation de temporisations avec `setitimer()`. Cet appel-système sert à fournir un suivi temporel de l'activité d'un processus. Il y a trois types de temporisations : la première fonctionne en temps réel et déclenche `SIGALRM` à son expiration, la deuxième ne fonctionne que lorsque le processus s'exécute et déclenche `SIGVTALRM`, et la troisième décompte le temps cumulé d'exécution du code du processus et celui du code du noyau exécuté lors des appels-système, puis déclenche `SIGPROF`. L'utilisation conjointe des deux dernières temporisations permet un suivi de l'activité du processus.

Il est donc formellement déconseillé d'utiliser simultanément les fonctions de comptabilité de `setitimer()` et la programmation de `alarm()`.

Notons de surcroît que l'implémentation de la fonction `sleep()` de la bibliothèque GlibC utilise également le signal `SIGALRM`. Cette fonction prend bien garde de ne pas interférer avec les éventuelles autres routines d'alarme du processus, sauf si le gestionnaire de `SIGALRM` installé par l'utilisateur se termine par un saut non local ; dans ce cas, elle échoue. Il peut arriver que, sur d'autres systèmes, la routine `sleep()` soit moins prévenante et qu'elle interfère avec `alarm()`. Pour une bonne portabilité d'un programme, il vaut donc mieux éviter d'utiliser les deux conjointement.

`SIGALRM` est défini par SUSv3. `SIGVTALRM` et `SIGPROF` également mais sous forme d'extension X/Open (pas toujours disponibles dans tous les Unix). Par défaut, ces trois signaux terminent le processus en cours.

Signaux SIGBUS et SIGSEGV

Ces deux signaux indiquent respectivement une erreur d'alignement des adresses sur le bus et une violation de la segmentation. Ils n'ont pas la même signification mais sont généralement dus au même type de bogue : l'emploi d'un pointeur mal initialisé.

Le signal SIGBUS est dépendant de l'architecture matérielle car il représente en fait une référence à une adresse mémoire invalide (par exemple un mauvais alignement de mots de deux ou quatre octets sur des adresses paires ou multiples de quatre).

Le signal SIGSEGV correspond à une adresse correcte mais pointant en dehors de l'espace d'adressage affecté au processus.

Ces deux signaux arrêtent par défaut le processus en créant un fichier core. Dans un cas comme dans l'autre, il est mal vu d'ignorer ces types de signaux, qui sont révélateurs de bogues. SUSv3 souligne d'ailleurs que le comportement d'un programme ignorant SIGSEGV est indéfini. À la rigueur, on peut capturer le signal, afficher un message à l'utilisateur et reprendre l'exécution dans un contexte propre par un saut non local longjmp(), comme nous le décrivons ci-dessous pour SIGILL.

Signaux SIGCHLD et SIGCLD

Le signal SIGCHLD est émis par le noyau vers un processus dont un fils vient de se terminer ou d'être stoppé. Ce processus peut alors soit ignorer le signal (ce qui est le comportement par défaut, mais qui est déconseillé par SUSv3), soit le capturer pour invoquer l'appel-système wait() qui précisera le PID du fils concerné et le code de terminaison s'il s'est fini.

Le signal SIGCLD est un synonyme de SIGCHLD sous Linux. Il s'agit d'une variante historique sur Système V. Le fonctionnement de SIGCLD était différent de celui de SIGCHLD et était particulièrement discutable. Les nouvelles applications doivent donc uniquement considérer SIGCHLD, qui est défini par SUSv3.

Par défaut, SIGCHLD est ignoré, mais ce comportement est légèrement différent de celui qui est adopté si on demande explicitement d'ignorer ce signal. Lorsqu'un processus fils se termine, et tant que son père n'a pas exécuté un appel wait(), il devient zombie si SIGCHLD est capturé par un gestionnaire ou s'il est traité par défaut. Par contre, le processus fils disparaît sans rester à l'état zombie si SIGCHLD est volontairement ignoré. SUSv3 déconseille toutefois qu'un processus ignore SIGCHLD s'il est destiné à avoir des descendants. Un gestionnaire de signaux minimal permettra d'éliminer facilement les zombies.

Signaux SIGFPE et SIGSTKFLT

SIGFPE correspond théoriquement à une « *Floating Point Exception* », mais il n'est en fait nullement limité aux erreurs de calcul en virgule flottante. Il inclut par exemple l'erreur de division entière par zéro. Il peut également se produire si le système ne possède pas de coprocesseur arithmétique, si le noyau a été compilé sans l'option pour l'émuler, et si le processus exécute des instructions mathématiques spécifiques.

SIGFPE est défini par Ansi C et SUSv3 ; par défaut, ce signal arrête le processus en créant un fichier core. Une application devra donc le laisser tel quel durant la phase de débogage afin de déterminer toutes les conditions dans lesquelles il se produit (en analysant post-mortem le fichier core). Théoriquement, un programme suffisamment défensif ne devrait laisser passer aucune condition susceptible de déclencher un signal SIGFPE, quelles que soient les données saisies par l'utilisateur…

On notera toutefois que le débogage peut parfois être malaisé en raison du retard du signal provenant du coprocesseur. Le signal n'est pas nécessairement délivré immédiatement après l'exécution de la condition d'erreur, mais peut survenir après plusieurs instructions.

SIGSTKFLT est un signal indiquant une erreur de pile, condition qui semble laisser les concepteurs de Linux particulièrement perplexes, comme en témoigne le commentaire dans /usr/include/signum.h :

```
#define SIGSTKFLT       16      /* ??? */
```

Ce signal, non conforme SUSv3, arrête par défaut le processus. On n'en trouve pas trace dans les sources du noyau, aussi est-il conseillé de ne pas s'en préoccuper.

Signal SIGHUP

Ce signal correspond habituellement à la déconnexion (*Hang Up*) du terminal de contrôle du processus. Le noyau envoie SIGHUP, suivi du signal SIGCONT que nous verrons plus bas, au processus leader de la session associée au terminal. Ce processus peut d'ailleurs se trouver en arrière-plan. La gestion de ce signal se trouve dans /usr/src/linux/drivers/char/tty_io.c.

SIGHUP est aussi envoyé, suivi de SIGCONT, à tous les processus d'un groupe qui devient orphelin. Rappelons qu'un processus crée un groupe en utilisant setpgid(), l'identifiant du groupe étant égal au PID du processus leader. Ses descendants futurs appartiendront automatiquement à ce nouveau groupe, à moins qu'ils ne créent le leur. Lorsque le processus leader se termine, le groupe est dit orphelin. À ce moment, le noyau envoie SIGHUP, suivi de SIGCONT si le groupe contient des processus arrêtés.

Enfin, il est courant d'envoyer manuellement le signal SIGHUP (en utilisant la commande /bin/kill) à certains processus démons, pour leur demander de se réinitialiser. Comme un démon n'a pas de terminal de contrôle, ce signal n'a pas de signification directe. Il est alors souvent utilisé pour demander au démon de relire ses fichiers de configuration (par exemple pour xinetd, named, sendmail…). En outre ces démons ferment puis rouvrent leurs fichiers de journalisation (*log*) à la réception de ce signal ce qui permet de réaliser une rotation de ces fichiers.

SIGHUP est décrit par SUSv3 et, par défaut, il termine le processus cible. On notera que la commande nohup permet de lancer un programme en l'immunisant contre SIGHUP. En fait, nohup n'est pas un programme C mais un script shell utilisant la fonctionnalité TRAP de l'interpréteur de commandes. Ce programme permet de lancer une application en arrière-plan, en redirigeant sa sortie vers le fichier nohup.out, et de se déconnecter en toute tranquillité.

Signal SIGILL

Ce signal est émis lorsque le processeur détecte une instruction assembleur illégale. Le noyau est prévenu par l'intermédiaire d'une interruption matérielle, et il envoie un signal SIGILL au processus fautif. Ceci ne doit jamais se produire dans un programme normal, compilé pour la bonne architecture matérielle. Mais il se peut toutefois que le fichier exécutable soit corrompu, ou qu'une erreur d'entrée-sortie ait eu lieu lors du chargement du programme et que le segment de code contienne effectivement un code d'instruction invalide.

Un autre problème possible peut être causé par les systèmes à base de 386 ne disposant pas de coprocesseur arithmétique. Le noyau utilise alors un émulateur (*wm-FPU-emu*) qui peut déclencher le signal SIGILL dans certains cas rares d'instructions mathématiques inconnues du 80486 de base.

En fait, l'occurrence la plus courante du signal SIGILL est révélatrice d'un bogue de débordement de pile. Par exemple, un tableau de caractères déclaré en variable locale (et donc alloué dans la pile) peut avoir été débordé par une instruction de copie de chaîne sans limite de longueur, comme strcpy(). La pile peut très bien être corrompue, et l'exécution du programme est totalement perturbée, avec une large confusion entre code et données.

Le comportement par défaut est d'arrêter le processus et de créer un fichier d'image mémoire core. Il s'agit d'un signal décrit dans la norme Ansi C. Il n'est d'aucune utilité d'ignorer ce signal. Un programme intelligemment conçu peut, en cas d'arrivée de SIGILL, adopter l'un des deux comportements suivants (dans le gestionnaire de signaux) :

- S'arrêter en utilisant exit(), pour éviter de laisser traîner un fichier core qui ne sera d'aucune aide à l'utilisateur, après avoir éventuellement affiché un message signalant la présence de bogue à l'auteur.
- Utiliser une instruction de saut non local longjmp(), que nous verrons un peu plus loin. Cette instruction permet de reprendre le programme à un point bien défini, avec un contexte propre. Il est toutefois préférable d'indiquer à l'utilisateur que l'exécution a été reprise à partir d'une condition anormale.

Signal SIGINT

Ce signal est émis vers tous les processus du groupe en avant-plan lors de la frappe d'une touche particulière du terminal : la touche d'interruption. Sur les claviers de PC sous Linux, ainsi que dans les terminaux Xterm, il s'agit habituellement de Contrôle-C. L'affectation de la commande d'interruption à la touche choisie peut être modifiée par l'intermédiaire de la commande stty, que nous détaillerons ultérieurement en étudiant les terminaux. L'affectation courante est indiquée au début de la deuxième ligne en invoquant stty -all :

```
$ stty --all
speed 9600 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
 [...]
$
```

SIGINT est défini par la norme Ansi C et SUSv3, et arrête par défaut le processus en cours.

Signaux SIGIO et SIGPOLL

Ces deux signaux sont synonymes sous Linux. SIGPOLL est le nom historique sous Système V, SIGIO celui de BSD.

SIGIO est envoyé lorsqu'un descripteur de fichier change d'état et permet la lecture ou l'écriture. Il s'agit généralement de descripteurs associés à des tubes, des sockets de connexion réseau, ou un terminal. La mise en place d'un gestionnaire de signaux pour SIGIO nécessite également la configuration du descripteur de fichiers pour accepter un fonctionnement asynchrone. Nous y reviendrons dans le chapitre 30, plus précisément dans les paragraphes consacrés aux entrées-sorties asynchrones.

L'utilisation de SIGIO permet à un programme d'accéder à un traitement d'entrée-sortie totalement asynchrone par rapport au déroulement du programme principal. Les données arrivant pourront servir par exemple à la mise à jour de variables globales consultées régulièrement dans le cours du programme normal. Il est alors important, lors de la lecture ou de l'écriture

de ces variables globales par le programme normal (en dehors du gestionnaire de signaux), de bien bloquer le signal `SIGIO` durant les modifications, comme nous apprendrons à le faire dans le reste de ce chapitre.

Un problème posé par les anciennes implémentations de `SIGIO` était l'impossibilité de déterminer automatiquement quel descripteur a déclenché le signal, dans le cas où plusieurs sont utilisés en mode asynchrone. Il fallait alors tenter systématiquement des lectures ou écritures non bloquantes sur chaque descripteur, en vérifiant celles qui échouent et celles qui réussissent. Ce défaut est corrigé depuis Linux 2.4.

Signal SIGKILL

`SIGKILL` est l'un des deux seuls signaux (avec `SIGSTOP`) qui ne puisse ni être capturé ni être ignoré, ni même être temporairement bloqué par un processus. À sa réception, tout processus est immédiatement arrêté. C'est une garantie pour s'assurer qu'on pourra toujours reprendre la main sur un programme particulièrement rétif. Le seul processus qui ne puisse pas recevoir `SIGKILL` est `init`, le processus de PID 1.

`SIGKILL` est traditionnellement associé à la valeur 9, d'où la célèbre ligne de commande `kill -9 xxx`, équivalente à `kill -KILL xxxx`. On notera que l'utilisation de `SIGKILL` doit être considérée comme un dernier recours, le processus ne pouvant se terminer proprement. On préfèrera essayer par exemple `SIGTERM` auparavant, puis `SIGQUIT` éventuellement.

Notons qu'un processus zombie n'est pas affecté par `SIGKILL` (ni par aucun autre signal d'ailleurs). Si un processus est stoppé, il sera relancé avant d'être terminé par `SIGKILL`.

Le noyau lui-même n'envoie que rarement ce signal. C'est le cas lorsqu'un processus a dépassé sa limite de temps d'exécution, ou lors d'un problème grave de manque de place mémoire. La bibliothèque GlibC n'utilise ce signal que pour tuer un processus qu'elle vient de créer lors d'un `popen()` et à qui elle n'arrive pas à allouer de flux de données.

Signal SIGPIPE

Ce signal est émis par le noyau lorsqu'un processus tente d'écrire dans un tube qui n'a pas de lecteur. Ce cas peut aussi se produire lorsqu'on tente d'envoyer des données dans une socket TCP/IP (traitée dans le chapitre sur la programmation réseau) dont le correspondant s'est déconnecté.

Le signal `SIGPIPE` (défini dans SUSv3) arrête par défaut le processus qui le reçoit. Toute application qui établit une connexion réseau doit donc soit intercepter le signal, soit (ce qui est préférable) l'ignorer. Dans ce dernier cas en effet, les appels système comme `write()` renverront une erreur `EPIPE` dans `errno`. On peut donc gérer les erreurs au coup par coup dès le retour de la fonction. Nous reviendrons sur ce problème dans les chapitres traitant des communications entre processus.

Signal SIGQUIT

Comme `SIGINT`, ce signal est émis par le pilote de terminal lors de la frappe d'une touche particulière : la touche `QUIT`. Celle-ci est affectée généralement à Contrôle-\ (ce qui nécessite sur les claviers français la séquence triple Ctrl-AltGr-\). `SIGQUIT` termine par défaut les processus qui ne l'ignorent pas et ne le capturent pas. `SIGQUIT` engendre en plus un fichier d'image mémoire (`core`).

Signaux SIGSTOP, SIGCONT, et SIGTSTP

`SIGSTOP` est le deuxième signal ne pouvant être ni capturé ni ignoré, comme `SIGKILL`. Il a toutefois un effet nettement moins dramatique que ce dernier, puisqu'il ne s'agit que d'arrêter temporairement le processus visé. Celui-ci passe à l'état *stoppé*.

Le signal `SIGCONT` a l'effet inverse : il permet de relancer un processus stoppé. Si le processus n'est pas stoppé ce signal n'a pas d'effet. Le redémarrage a toujours lieu, même si `SIGCONT` est capturé par un gestionnaire de signaux de l'utilisateur ou s'il est ignoré.

`SIGTSTP` est un signal ayant le même effet que `SIGSTOP`, mais il peut être capturé ou ignoré, et il est émis par le terminal de contrôle vers le processus en avant-plan. Lors d'un `stty --all`, la touche affectée à ce signal est indiquée par `susp=` et non par `stop=` qui correspond à un arrêt temporaire de l'affichage sur le terminal. Dans la plupart des cas, il s'agit de la touche Contrôle-Z.

Il est rare qu'une application ait besoin de capturer les signaux `SIGTSTP` et `SIGCONT`, mais cela peut arriver si elle doit gérer le terminal de manière particulière (par exemple en affichant des menus déroulants), et si elle désire effacer l'écran lorsqu'on la stoppe et le redessiner lorsqu'elle redémarre.

Dans la plupart des cas, on ne s'occupera pas du comportement de ces signaux.

Signal SIGTERM

Ce signal est une demande « gentille » de terminaison d'un processus. Il peut être ignoré ou capturé pour terminer proprement le programme en ayant effectué toutes les tâches de nettoyage nécessaires. Traditionnellement numéroté 15, ce signal est celui qui est envoyé par défaut par la commande `/bin/kill`.

`SIGTERM` est défini par Ansi C et SUSv3. Par défaut, il termine le processus concerné. En principe, une application bien conçue devrait installer systématiquement un gestionnaire pour `SIGTERM` et `SIGINT` afin d'assurer une fin « propre » au programme lorsque l'utilisateur a besoin de l'arrêter rapidement.

Signal SIGTRAP

Ce signal est émis par le noyau lorsque le processus a atteint un point d'arrêt. `SIGTRAP` est utilisé par les débogueurs comme `gdb`. Il n'a pas d'intérêt pour une application classique. Le comportement par défaut d'un processus recevant `SIGTRAP` est de s'arrêter avec un fichier `core`. On peut à la rigueur l'ignorer dans une application qui n'a plus besoin d'être déboguée et qui ne désire pas créer de fichier `core` intempestif.

Signaux SIGTTIN et SIGTTOU

Ces signaux sont émis par le terminal en direction d'un processus en arrière-plan, qui essaye respectivement de lire depuis le terminal ou d'écrire dessus. Lorsqu'un processus en arrière-plan tente de lire depuis son terminal de contrôle, tous les processus de son groupe reçoivent le signal `SIGTTIN`. Par défaut, cela stoppe les processus (sans les terminer). Toutefois, il est possible d'ignorer ce signal ou de le capturer. Dans ce cas, l'appel système de lecture `read()` échoue et renvoie un code d'erreur `EIO` dans `errno`.

Un phénomène assez semblable se produit dans le cas où un processus essaye d'écrire sur son terminal de contrôle alors qu'il se trouve en arrière-plan. Tous les processus du groupe reçoivent `SIGTTOU`, qui les stoppe par défaut. Si ce signal est ignoré ou capturé, l'écriture a quand même lieu. Notons que l'interdiction d'écrire sur le terminal par un processus en arrière-plan n'est valable que si l'attribut `TOSTOP` du terminal est actif (ce qui n'est pas le cas par défaut).

Nous voyons ci-dessous l'exemple avec le processus `/bin/ls` qu'on invoque en arrière-plan. Par défaut, il écrit quand même sur le terminal. Lorsqu'on active l'attribut `TOSTOP` du terminal, `/bin/ls` est automatiquement stoppé quand il tente d'écrire. On peut le relancer en le ramenant en avant-plan avec la commande `fg` du shell.

```
$ ls &
[1] 2839
$ bin     etc     lost+found  root    usr
boot    home    mnt         sbin    var
dev     lib     proc        tmp
[1]+  Done              ls
$ stty tostop
$ stty
speed 9600 baud; line = 0;
-brkint -imaxbel
tostop
$ ls &
[1] 2842
$

[1]+  Stopped (tty output)    ls
$ fg
ls
bin       etc       lost+found  root      usr
boot      home      mnt         sbin      var
dev       lib       proc        tmp
$
```

Signal SIGURG

Ce signal est émis par le noyau lorsque des données « hors bande » arrivent sur une connexion réseau. Ces données peuvent être transmises sur une socket TCP/IP avec une priorité plus grande que pour les données normales. Le processus récepteur est alors averti par ce signal de leur arrivée. Le comportement par défaut est d'ignorer ce signal. Seule une application utilisant un protocole réseau assez complexe pourra avoir besoin d'intercepter ce signal. Notons qu'une telle application peut détecter également l'arrivée de données hors bande différemment, par le biais de l'appel système `select()`.

Signaux SIGUSR1 et SIGUSR2

Ces deux signaux sont à la disposition du programmeur pour ses applications. Le fait que seuls deux signaux soient réservés à l'utilisateur peut sembler restreint, mais cela permet déjà d'établir un système – minimal – de communications interprocessus.

Ces deux signaux sont définis par SUSv3. Par défaut, ils terminent le processus visé, ce qui peut sembler discutable.

Signal SIGWINCH

SIGWINCH (*Window changed*) indique que la taille du terminal a été modifiée. Ce signal, ignoré par défaut, est principalement utile aux applications se déroulant en plein écran texte. Lorsque ces dernières sont exécutées dans un Xterm, il est plus sympathique qu'elles se reconfigurent automatiquement lorsqu'on modifie la taille de la fenêtre du Xterm.

C'est par exemple le cas de l'utilitaire less, entre autres, qui intercepte SIGWINCH pour recalculer ses limites d'écran et rafraîchir son affichage.

Une application interactive plein écran gérant ce signal bénéficie d'une qualité non négligeable en termes d'ergonomie utilisateur. C'est un peu le premier degré du fonctionnement dans un environnement graphique multifenêtré.

Par contre, la modification asynchrone des caractéristiques d'affichage peut parfois être difficile à implémenter au niveau programmation. La méthode la plus simple consiste à modifier un drapeau placé dans une variable globale, qui sera consultée régulièrement dans la boucle principale de fonctionnement du programme, là où des modifications de taille d'écran peuvent plus aisément être prises en compte.

Signaux SIGXCPU et SIGXFSZ

Ces signaux sont émis par le noyau lorsqu'un processus dépasse une de ses limites souples des ressources système. Nous verrons en détail ultérieurement, avec les appels système getrlimit() et setrlimit(), que chaque processus est soumis à plusieurs types de limites système (par exemple le temps d'utilisation du CPU, la taille maximale d'un fichier ou de la zone de données, le nombre maximal de fichiers ouverts…). Pour chaque limite, il existe une valeur souple, modifiable par l'utilisateur, et une valeur stricte, modifiable uniquement par *root* ou par un processus ayant la capacité CAP_SYS_RESOURCE.

Lorsqu'un processus a dépassé sa limite souple d'utilisation du temps processeur, il reçoit chaque seconde le signal SIGXCPU. Celui-ci arrête le processus par défaut, mais il peut être ignoré ou capturé. Lorsque le processus tente de dépasser sa limite stricte, le noyau le tue avec un signal SIGKILL.

SIGXFSZ fonctionne de manière similaire et est émis par le noyau lorsqu'un processus tente de créer un fichier trop grand. Le signal n'est émis toutefois que si l'appel système write() tente de dépasser en une seule fois la taille maximale. Dans les autres cas, l'écriture au-delà du seuil programmé déclenche une erreur EFBIG de l'appel write().

Signaux temps-réel

Les signaux temps-réel représentent une extension importante apparue entre le noyau Linux 2.2, car ils donnent accès aux fonctionnalités définies à l'époque par la norme Posix.1b. De nos jours, ils sont décrits dans l'extension RTS (*real time signals*) de SUSv3. Pour vérifier l'existence de ces signaux en cas de portage d'une application, on peut tester à la compilation la présence de la constante symbolique **_POSIX_REALTIME_SIGNALS** dans le fichier <unistd.h>.

Les signaux temps-réel sont réservés à l'utilisateur : ils ne sont pas déclenchés par des événements détectés par le noyau. Ce sont donc des extensions de SIGUSR1 et SIGUSR2. Ils ne sont pas représentés par des constantes, mais directement par leurs valeurs, qui s'étendent de SIGRTMIN à SIGRTMAX, bornes incluses. Ces signaux sont particuliers – nous y reviendrons – car ils sont ordonnés en fonction de leur priorité, empilés à la réception (donc plusieurs signaux

du même type peuvent être reçus très rapidement), et ils fournissent au processus des informations plus précises que les autres signaux. Nous parlerons des fonctionnalités temps-réel dans le chapitre 8.

Voici donc l'essentiel des signaux gérés par Linux. Il existe quelques signaux inutilisés dans la plupart des cas (`SIGLOST`, qui surveille les verrous sur les fichiers NFS), et d'autres dont le comportement diffère suivant l'architecture matérielle (`SIGPWR` / `SIGINFO`, qui préviennent d'une chute d'alimentation électrique).

Pour obtenir le libellé d'un signal dont le numéro est connu (comme nous le ferons par la suite dans un gestionnaire), il existe plusieurs méthodes :

- La fonction **`strsignal()`** disponible en tant qu'extension Gnu (il faut donc utiliser la constante symbolique `_GNU_SOURCE` à la compilation) dans `<string.h>`. Cette fonction renvoie un pointeur sur une chaîne de caractères allouée statiquement (donc susceptible d'être écrasée à chaque appel) et contenant un libellé descriptif du signal.
- La fonction **`psignal()`** affiche sur la sortie d'erreur standard la chaîne passée en second argument (si elle n'est pas `NULL`), suivie d'un deux points et du descriptif du signal dont le numéro est fourni en premier argument. Cette fonction est plus portable que `strsignal()`, elle est définie dans `<signal.h>`.
- Il existe une table globale de chaînes de caractères contenant les libellés des signaux : `char * sys_siglist [numero_signal]`.

Les prototypes de `strsignal()` et de `psignal()` sont les suivants :

```
char * strsignal (int numero_signal);

int psignal (int numero_signal, const char * prefixe_affiche);
```

Voici un exemple permettant de consulter les libellés.

exemple_strsignal.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <string.h>

    int
main (void)
{
    int i;

    fprintf(stderr, "strsignal() :\n");
    for (i = 1; i < NSIG; i ++)
        fprintf(stderr, "%s\n", strsignal(i));

    fprintf(stderr, "\n sys_siglist[] : \n");
    for (i = 1; i < NSIG; i ++)
        fprintf (stderr, "%d : %s\n", i, sys_siglist[i]);

    return EXIT_SUCCESS;
}
```

Dans l'exemple d'exécution ci-dessous, on remarque que `strsignal()` est sensible à la localisation en cours (fr_FR). Ce n'est pas le cas pour `sys_siglist[]`.

```
$ ./exemple_strsignal
strsignal() :
Fin de la connexion (raccroché)
Interruption
Quitter
Instruction illégale
Trappe pour point d'arrêt et de trace
Abandon
Erreur du bus
Exception en point flottant
Processus arrêté
Signal #1 défini par l'usager
Erreur de segmentation
Signal #2 défini par l'usager
Relais brisé (pipe)
Minuterie d'alerte
Complété
Erreur sur la pile
Le processus `enfant' a terminé.
Poursuite
Signal d'arrêt
Arrêté
Arrêté (via l'entrée sur tty)
Arrêté (via la sortie sur tty)
Condition d'E/S urgente
Temps UCT limite expiré
Débordement de la taille permise pour un fichier
Expiration de la minuterie virtuelle
Expiration de la minuterie durant l'établissement du profil
La fenêtre a changé.
E/S possible
Panne d'alimentation
Signal inconnu 31
Signal de Temps-Réel 0
Signal de Temps-Réel 1
Signal de Temps-Réel 2
  [...]
Signal de Temps-Réel 29
Signal de Temps-Réel 30
Signal de Temps-Réel 31

sys_siglist[] :
1 : Hangup
2 : Interrupt
3 : Quit
4 : Illegal instruction
5 : Trace/breakpoint trap
6 : Aborted
7 : Bus error
```

```
8 : Floating point exception
9 : Killed
10 : User defined signal 1
11 : Segmentation fault
12 : User defined signal 2
13 : Broken pipe
14 : Alarm clock
15 : Terminated
16 : Stack fault
17 : Child exited
18 : Continued
19 : Stopped (signal)
20 : Stopped
21 : Stopped (tty input)
22 : Stopped (tty output)
23 : Urgent I/O condition
24 : CPU time limit exceeded
25 : File size limit exceeded
26 : Virtual timer expired
27 : Profiling timer expired
28 : Window changed
29 : I/O possible
30 : Power failure
31 : (null)
32 : (null)
33 : (null)
   [...]
60 : (null)
61 : (null)
62 : (null)
63 : (null)
$
```

Nous allons à présent étudier comment un signal est envoyé à un processus, et sous quelles conditions il lui parvient. Pour l'instant, nous parlerons des signaux classiques et nous réserverons les modifications concernant les signaux temps-réel au chapitre 8.

Émission d'un signal sous Linux

Pour envoyer un signal à un processus, on utilise l'appel système `kill()`, qui est particulièrement mal nommé car il ne tue que rarement l'application cible. Cet appel système est déclaré ainsi :

```
int kill (pid_t pid, int numero_signal);
```

Le premier argument correspond au PID du processus visé, et le second au numéro du signal à envoyer. Rappelons qu'il est essentiel d'utiliser la constante symbolique correspondant au nom du signal et non la valeur numérique directe.

Il existe une application `/bin/kill` qui sert de frontal en ligne de commande à l'appel système `kill()`. `/bin/kill` prend en option sur la ligne de commande un numéro de signal précédé d'un tiret « - » et suivi du ou des PID des processus à tuer. Le numéro de signal peut être

remplacé par le nom symbolique du signal, avec ou sans le préfixe `SIG`. Par exemple, sur Linux version x86, `kill -13 1234` est équivalent à `kill -PIPE 1234`. Si on ne précise pas de numéro de signal, `/bin/kill` utilise `SIGTERM` par défaut.

En fait, le premier argument de l'appel système `kill()` peut prendre diverses valeurs :

- S'il est strictement positif, le signal est envoyé au processus dont le PID correspond à cet argument.
- S'il est nul, le signal est envoyé à tous les processus du groupe auquel appartient le processus appelant.
- S'il est négatif, sauf pour –1, le signal est envoyé à tous les processus du groupe dont le PGID est égal à la valeur absolue de cet argument.
- S'il vaut –1, SUSv3 indique que le comportement est indéfini. Sous Linux, le signal est envoyé à tous les processus sur le système, sauf au processus `init` (PID=1) et au processus appelant. Cette option est utilisée, par exemple dans les scripts de `shutdown` pour tuer les processus avant de démonter les systèmes de fichiers et d'arrêter le système (par exemple sur une distribution type Red Hat, on trouve ceci dans `/etc/rc.d/rc0.d/S01halt`).

Bien entendu, l'émission du signal est soumise à des contraintes d'autorisation :

- Tout d'abord, un processus ayant son UID effectif nul ou la capacité `CAP_KILL` peut envoyer des signaux à tout processus.
- Un processus peut envoyer un signal à un autre processus si l'UID réel ou effectif de l'émetteur est égal à l'UID réel ou sauvé de la cible.
- Enfin, si le signal est `SIGCONT`, il suffit que le processus émetteur appartienne à la même session que le processus visé.

Dans tous les autres cas, l'appel système `kill()` échoue et remplit `errno` avec le code `EPERM`. Si toutefois aucun processus ne correspond au premier argument – quelles que soient les autorisations –, `errno` contiendra `ESRCH`. Le second argument de `kill()`, le numéro du signal à envoyer, doit être positif ou nul, et inférieur à `NSIG`, sinon `kill()` renvoie l'erreur `EINVAL`.

Nous voyons ici l'intérêt du signal numéro 0. Il ne s'agit pas vraiment d'un signal – rien n'est émis –, mais il permet de savoir si un processus est présent sur le système. Si c'est le cas, l'appel système `kill()` réussira si on peut atteindre le processus par un signal, ou il échouera avec `EPERM` si on n'en a pas le droit. Si le processus n'existe pas, il échouera avec `ESRCH`. Voici par exemple un moyen de l'employer :

```
    int
processus_present (pid_t pid)
{
    if (kill(pid, 0) == 0)
        return VRAI;
    if (errno == EPERM)
        return VRAI;
    return FAUX;
}
```

Bien entendu, il est toujours possible qu'un processus qui existait encore lors de l'appel `kill(pid, 0)` se soit terminé avant le retour de cette fonction, ou que l'effet inverse se produise.

Il existe une fonction de la bibliothèque C nommée **raise()**, et déclarée ainsi :

```
void raise (int numero_signal);
```

Elle est équivalente à `kill(getpid(), numero_signal)` et permet de s'envoyer un signal. Cette fonction est incluse dans la norme Ansi C car elle ne nécessite pas de notion de PID, à la différence de `kill()` qui elle est définie par SUSv3.

Un processus qui s'envoie un signal à lui-même, que ce soit par `raise(num)` ou par `kill(getpid(), num)`, est assuré que le signal sera délivré avant le retour de l'appel système `kill()`. Si le signal est capturé, le gestionnaire sera exécuté dans tous les cas avant le retour de `kill()`. Ce n'est bien entendu pas le cas lorsque le signal est destiné à un autre processus. Le signal est mis en attente (suspendu) jusqu'à ce qu'il soit effectivement délivré au processus visé.

Il existe également une fonction de bibliothèque nommée **killpg()** permettant d'envoyer un signal à un groupe de processus. Elle est déclarée ainsi :

```
int killpg (pid_t pgid, int numero_signal)
```

Si le `pgid` indiqué est nul, elle envoie le signal au groupe du processus appelant. Comme on pouvait s'y attendre, elle fait directement appel à `kill(-pgid, numero_signal)` après avoir vérifié que `pgid` est positif ou nul. Un tel appel système est intéressant car il peut détruire automatiquement tous ses descendants en se terminant. Pour cela, il suffit que le processus initial crée son propre groupe en invoquant `setpgrp()`, et il pourra tuer tous ses descendants en appelant `killpg(getpgrp(), SIGKILL)` avant de se finir.

L'émission des signaux est, nous le voyons, une chose assez simple et portable. Nous allons observer comment le noyau se comporte pour délivrer un signal en fonction de l'action programmée par le processus.

Délivrance des signaux – Appels système lents

Un processus peut bloquer temporairement un signal. Si celui-ci arrive pendant ce temps, il reste en attente jusqu'à ce que le processus le débloque. Il est bien sûr impossible de bloquer `SIGKILL` ou `SIGSTOP`. Le fait de bloquer un signal permet de protéger des parties critiques du code, par exemple l'accès à une variable globale modifiée également par le gestionnaire du signal en question.

Les concepteurs de Linux 2.0 avaient choisi de représenter la liste des signaux en attente pour une tâche par un masque de bits contenu dans un entier long. Chaque signal correspondait donc à un bit précis. Depuis Linux 2.2 a été introduit le concept des signaux temps-réel, pour lesquels il existe une file d'attente des signaux. Un même signal temps-réel peut donc être en attente en plusieurs occurrences. La représentation interne des signaux en attente a donc été modifiée pour devenir une table.

Un processus possède également un masque de bits indiquant quels sont les signaux bloqués. Lorsqu'un processus reçoit un signal bloqué, ce dernier reste en attente jusqu'à ce que le processus le débloque. Les signaux classiques ne sont pas empilés, ce qui signifie qu'une seule occurrence d'un signal donné est conservée lorsque plusieurs exemplaires arrivent consécutivement alors que le signal est bloqué. Les signaux temps-réel sont par contre empilés, ce qui veut dire que le noyau conserve l'ensemble des exemplaires d'un signal arrivé alors qu'il est bloqué.

Le travail du noyau commence donc à partir de la fonction `kill()` interne se trouvant dans `/usr/src/linux/kernel/signal.c`. Le noyau analyse le masque des signaux bloqués du processus visé. Si le signal est bloqué, il est donc simplement inscrit dans la liste en attente. Si le signal n'est pas bloqué et si le processus cible est interruptible, ce dernier est alors réveillé.

À son réveil, le processus traite la liste des signaux en attente non bloqués. Un signal ignoré est simplement éliminé, sauf `SIGCHLD`, pour lequel le système exécute en plus une boucle

```
while (waitpid(-1, NULL, WNOHANG) > 0)
    /* rien */;
```

pour éliminer les zombies.

Le système assure ensuite la gestion des signaux dont le comportement est celui par défaut (ignorer, stopper, arrêter, créer un fichier `core`). Sinon, il invoque le gestionnaire de signaux installé par l'utilisateur.

Il existe sous Unix des appels système rapides et des appels système lents. Un appel rapide ne peut pas être interrompu (hormis par une commutation de tâche de l'ordonnanceur). Par contre, un appel lent peut rester bloqué pendant une durée indéterminée. Savoir si un appel système est rapide ou lent n'est pas toujours simple. Classiquement, tous les appels concernant des descripteurs de fichiers (`open`, `read`, `write`, `fcntl`…) peuvent être lents dès lors qu'ils travaillent sur une socket réseau, un tube, voire des descripteurs de terminaux. Bien entendu, les appels système d'attente comme `wait()`, `select()`, `poll()` ou `pause()` peuvent attendre indéfiniment. Certains appels système servant aux communications interprocessus, comme `semop()` qui gère des sémaphores ou `msgsnd()` et `msgrcv()` qui permettent de transmettre des messages, peuvent rester bloqués au gré du processus correspondant.

Prenons l'exemple d'une lecture depuis une connexion réseau. L'appel système `read()` est alors bloqué aussi longtemps que les données se font attendre. Si un signal non bloqué, pour lequel un gestionnaire a été installé, arrive pendant un appel système lent, ce dernier est interrompu. Le processus exécute alors le gestionnaire de signaux. À la fin de l'exécution de celui-ci (dans le cas où il n'a pas mis fin au programme ni exécuté de saut non local), il y a plusieurs possibilités.

Le noyau peut faire échouer l'appel interrompu, qui transmet alors le code d'erreur `EINTR` dans `errno`. Le programme utilisateur devra alors réessayer son appel. Ceci implique d'encadrer tous les appels système lents avec une gestion du type :

```
do {
    nb_lus = read(socket_rcpt, buffer, nb_octets_a_lire);
} (while ((nb_lus == -1) && (errno == EINTR));
```

Ceci est tout à fait utilisable si, dans les portions de code où des signaux sont susceptibles de se produire, on utilise peu d'appels système lents. Notons d'ailleurs que les fonctions de la bibliothèque C, par exemple `fread()`, gèrent elles-mêmes ce genre de cas.

En outre, le fait de faire volontairement échouer une lecture est un moyen d'éviter un blocage définitif, en utilisant un délai maximal par exemple. L'appel système `alarm()`, qui déclenche un signal `SIGALRM` lorsque le délai prévu est écoulé, est bien sûr le plus couramment utilisé.

```
alarm(delai_maximal_en_secondes);
nb_lus = read(socket_rcpt, buffer, nb_octets_a_lire);
alarm(0); /* arrêter la tempo si pas écoulée entièrement */
```

```
    if (nb_lus != nb_octets_a_lire) {
        if (errno == EINTR)
            /* Délai dépassé... */
```

Ce code est très imparfait ; nous en verrons d'autres versions quand nous étudierons plus en détail l'alarme. De plus, nous ne savons pas quel signal a interrompu l'appel système, ce n'est pas nécessairement `SIGALRM`.

Il faut alors tester tous les retours de fonctions du type `read()` ou `write()`, et les relancer éventuellement si le signal reçu n'a pas d'influence sur le travail en cours. C'est d'autant plus contraignant avec le développement des applications fonctionnant en réseau, où une grande partie des appels système autrefois rapides – `read()` depuis un fichier sur disque – peuvent bloquer longuement, le temps d'interroger un serveur distant. La surcharge en termes de code nécessaire pour encadrer tous les appels système susceptibles de bloquer est parfois assez lourde.

Une autre possibilité, introduite initialement par les systèmes BSD, est de demander au noyau de relancer automatiquement les appels système interrompus. Ainsi, le code utilisant `read()` ne se rendra pas compte de l'arrivée du signal, le noyau ayant fait redémarrer l'appel système comme si de rien n'était. L'appel `read()` ne renverra jamais plus l'erreur `EINTR`.

Cela peut se configurer aisément, signal par signal. Il est donc possible de demander que tous les signaux pour lesquels un gestionnaire est fourni fassent redémarrer automatiquement les appels système interrompus, à l'exception par exemple de `SIGALRM` qui peut servir à programmer un délai maximal. Dans l'exemple précédent, la lecture reprendra automatiquement et ne se terminera que sur une réussite ou une réelle condition d'erreur, sauf bien entendu si on la temporise avec `alarm()`.

Nous allons à présent étudier le moyen de configurer le comportement d'un processus à la réception d'un signal précis.

Réception des signaux avec l'appel système signal()

Un processus peut demander au noyau d'installer un gestionnaire pour un signal particulier, c'est-à-dire une routine spécifique qui sera invoquée lors de l'arrivée de ce signal. Le processus peut aussi vouloir que le signal soit ignoré lorsqu'il arrive, ou laisser le noyau appliquer le comportement par défaut (souvent une terminaison du programme).

Pour indiquer son choix au noyau, il y a deux possibilités :

- L'appel-système `signal()`, défini par Ansi C et SUSv3, présente l'avantage d'être très simple (on installe un gestionnaire en une seule ligne de code), mais il peut parfois poser des problèmes de fiabilité de délivrance des signaux et de compatibilité entre les divers systèmes Unix.
- L'appel-système `sigaction()` est légèrement plus complexe puisqu'il implique le remplissage d'une structure, mais il permet de définir précisément le comportement désiré pour le gestionnaire, sans ambiguïté suivant les systèmes puisqu'il est complètement défini par SUSv3.

Nous allons tout d'abord voir la syntaxe et l'utilisation de `signal()`, car il est souvent employé, puis nous étudierons dans le prochain chapitre `sigaction()`, qui est généralement plus adéquat pour contrôler finement le comportement d'un programme. Notons au passage

l'existence d'une ancienne fonction, **sigvec()**, obsolète de nos jours et approximativement équivalente à sigaction().

L'appel système **signal()** présente un prototype qui surprend toujours au premier coup d'œil, alors qu'il est extrêmement simple en réalité :

```
void (*signal (int numero_signal, void (*gestionnaire) (int))) (int);
```

Il suffit en fait de le décomposer, en créant un type intermédiaire correspondant à un pointeur sur une routine de gestion de signaux :

```
typedef void (*gestion_t)(int);
```

et le prototype de signal() devient :

```
gestion_t signal (int numero_signal, gestion_t gestionnaire);
```

En d'autres termes, signal() prend en premier argument un numéro de signal. Bien entendu, il faut utiliser la constante symbolique correspondant au nom du signal, jamais la valeur numérique directe. Le second argument est un pointeur sur la routine qu'on désire installer comme gestionnaire de signaux. L'appel système nous renvoie un pointeur sur l'ancien gestionnaire, ce qui permet de le sauvegarder pour éventuellement le réinstaller plus tard.

Il existe deux constantes symboliques qui peuvent remplacer le pointeur sur un gestionnaire, **SIG_IGN** et **SIG_DFL**, qui sont définies dans <signal.h>.

La constante SIG_IGN demande au noyau d'ignorer le signal indiqué. Par exemple l'appel système signal(SIGCHLD, SIG_IGN) – déconseillé par SUSv3 – a ainsi pour effet sous Linux d'éliminer directement les processus fils qui se terminent, sans les laisser à l'état zombie.

Avec la constante SIG_DFL, on demande au noyau de réinstaller le comportement par défaut pour le signal considéré. Nous avons vu l'essentiel des actions par défaut. Elles sont également documentées dans la page de manuel signal(7).

Si l'appel système signal() échoue, il renvoie une valeur particulière, elle aussi est définie dans <signal.h> : **SIG_ERR**.

L'erreur positionnée dans errno est alors généralement EINVAL, qui indique un numéro de signal inexistant. Si on essaie d'ignorer ou d'installer un gestionnaire pour les signaux SIGKILL ou SIGSTOP, l'opération n'a pas lieu. La documentation de la fonction signal() de GlibC indique que la modification est silencieusement ignorée, mais en réalité l'appel système sigaction() – interne au noyau –, sur lequel cette fonction est bâtie, renvoie EINVAL dans errno dans ce cas.

L'erreur EFAULT peut aussi être renvoyée dans errno si le pointeur de gestionnaire de signaux n'est pas valide.

Un gestionnaire de signaux est une routine comme les autres, qui prend un argument de type entier et qui ne renvoie rien. L'argument transmis correspond au numéro du signal ayant déclenché le gestionnaire. Il est donc possible d'écrire un unique gestionnaire pour plusieurs signaux, en répartissant les actions à l'aide d'une construction switch-case.

Il arrive que le gestionnaire de signaux puisse recevoir d'autres informations dans une structure transmise en argument supplémentaire (comme le PID du processus ayant envoyé le signal). Cette fonctionnalité est disponible depuis Linux 2.2. Pour cela, il faut installer nécessairement le gestionnaire avec l'appel système sigaction() que nous verrons plus bas.

Le gestionnaire de signaux étant une routine sans spécificité, il est possible de l'invoquer directement dans le corps du programme si le besoin s'en fait sentir.

Nous allons pouvoir installer notre premier gestionnaire de signaux. Nous allons tenter de capturer tous les signaux. Bien entendu, `signal()` échouera pour `SIGKILL` et `SIGSTOP`. Pour tous les autres signaux, notre programme affichera le PID du processus en cours, suivi du numéro de signal et de son nom. Il faudra disposer d'une seconde console (ou d'un autre Xterm) pour pouvoir tuer le processus à la fin.

exemple_signal.c :

```
     #include <stdio.h>
     #include <stdlib.h>
     #include <signal.h>
     #include <unistd.h>

    void
gestionnaire (int numero_signal)
{
    fprintf(stdout, "\n %ld a reçu le signal %d (%s)\n",
         (long) getpid(), numero_signal, sys_siglist[numero_signal]);
}

    int
main (void)
{
    int i;

    for (i = 1; i < NSIG; i ++)
        if (signal(i, gestionnaire) == SIG_ERR)
            fprintf(stderr, "Signal %d non capturé \n", i);
    while (1)
        pause();

    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution avec, en seconde colonne, l'action effectuée sur un autre terminal :

```
$ ./exemple_signal
Signal 9 non capturé
Signal 19 non capturé
  (Contrôle-C)
6240 a reçu le signal 2 (Interrupt)
  (Contrôle-Z)
6240 a reçu le signal 20 (Stopped)
  (Contrôle-\)
6240 a reçu le signal 3 (Quit)
                                    $ kill -TERM 6240
6240 a reçu le signal 15 (Terminated)
                                    $ kill -KILL 6240
Killed
$
```

On appuie sur les touches de contrôle sur la console du processus `exemple_signal`, alors que les ordres `kill` sont envoyés depuis une autre console. Le signal 9 non capturé correspond à `SIGKILL`, et le 19 à `SIGSTOP`.

Ce programme a également un comportement intéressant vis-à-vis du signal `SIGSTOP`, qui le stoppe temporairement. Le shell reprend alors la main. Nous pouvons toutefois ramener le processus en avant-plan, ce qui lui transmet le signal `SIGCONT` :

```
$ ./exemple_signal
Signal 9 non capturé
Signal 19 non capturé
  (Contrôle-C)
6241 a reçu le signal 2 (Interrupt)
                                        $ kill -STOP 6241
[1]+  Stopped (signal)    ./exemple_signal
$ ps 6241
  PID TTY STAT  TIME COMMAND
 6241  p5 T    0:00 ./exemple_signal
$ fg
./exemple_signal
6241 a reçu le signal 18 (Continued)
  (Contrôle-\)
6241 a reçu le signal 3 (Quit)
                                        $ kill -KILL 6241
Killed
$
```

Le champ STAT de la commande `ps` contient `T`, ce qui correspond à un processus stoppé ou suivi (*traced*).

Il faut savoir que sous Linux, la constante symbolique `SIG_DFL` est définie comme valant 0 (c'est souvent le cas, même sur d'autres systèmes Unix). Lors de la première installation d'un gestionnaire, l'appel système `signal()` renvoie donc, la plupart du temps, cette valeur (à moins que le shell n'ait modifié le comportement des signaux auparavant). Il y a là un risque d'erreur pour le programmeur distrait qui peut écrire machinalement :

```
    if (signal(...) != 0)
        /* erreur */
```

comme on a l'habitude de le faire pour d'autres appels système. Ce code fonctionnera à la première invocation, mais échouera par la suite puisque `signal()` renvoie l'adresse de l'ancien gestionnaire. Ne pas oublier, donc, de détecter les erreurs ainsi :

```
    if (signal(...) == SIG_ERR)
        /* erreur */
```

Nous avons pris soin dans l'exécution de l'exemple précédent de ne pas invoquer deux fois de suite le même signal. Pourtant, cela n'aurait pas posé de problème avec Linux et la GlibC, comme en témoigne l'essai suivant :

```
$ ./exemple_signal
Signal 9 non capturé
Signal 19 non capturé
  (Contrôle-C)
6743 a reçu le signal 2 (Interrupt)
```

```
  (Contrôle-C)
6743 a reçu le signal 2 (Interrupt)
  (Contrôle-C)
6743 a reçu le signal 2 (Interrupt)
  (Contrôle-Z)
6743 a reçu le signal 20 (Stopped)
  (Contrôle-Z)
6743 a reçu le signal 20 (Stopped)
                                        $ kill -KILL 6743
Killed
$
```

Il existe toutefois de nombreux systèmes Unix (de la famille Système V) sur lesquels un gestionnaire de signaux ne reste pas en place après avoir été invoqué. Une fois que le signal est arrivé, le noyau repositionne le comportement par défaut. Ce dernier peut être observé sous Linux avec GlibC en définissant la constante symbolique `_XOPEN_SOURCE` avant d'inclure `<signal.h>`. En voici un exemple :

exemple_signal_2.c :

```
    #define _XOPEN_SOURCE
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>

    void
gestionnaire (int numero_signal)
{
    fprintf(stdout, "\n %ld a reçu le signal %d\n", (long) getpid (),
             numero_signal);
}

    int
main (void)
{
    int i;

    for (i = 1; i < _NSIG; i ++)
        if (signal(i, gestionnaire) == SIG_ERR)
            fprintf(stderr, "%ld ne peut capturer le signal %d\n",
                    (long) getpid(), i);
    while (1)
        pause();

    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution dans lequel on remarque que la première frappe de Contrôle-Z est interceptée, mais pas la seconde, qui stoppe le processus et redonne la main au shell. On redémarre alors le programme avec la commande `fg`, et on invoque Contrôle-C. Sa première occurrence sera bien interceptée, mais pas la seconde.

```
$ ./exemple_signal_2
6745 ne peut capturer le signal 9
6745 ne peut capturer le signal 19
  (Contrôle-Z)
6745 a reçu le signal 20
  (Contrôle-Z)
[1]+  Stopped       ./exemple_signal_2
$ ps 6745
  PID TTY STAT  TIME COMMAND
 6745  p5 T    0:00 ./exemple_signal_2
$ fg
./exemple_signal_2

6745 a reçu le signal 18
  (Contrôle-C)
6745 a reçu le signal 2
  (Contrôle-C)
$
```

Le signal 18 correspond à `SIGCONT`, que le shell a envoyé en replaçant le processus en avant-plan. Sur ce type de système, il est nécessaire que le gestionnaire de signaux s'installe à nouveau à chaque interception d'un signal. On doit donc utiliser un code du type :

```
    int
gestionnaire (int numero_signal)
{
    signal(numero_signal, gestionnaire);
    /* Traitement effectif du signal reçu */
}
```

Il est toutefois possible que le signal arrive de nouveau avant que le gestionnaire ne soit réinstallé. Ce type de comportement à risque conduit à avoir des signaux non fiables.

Un deuxième problème se pose avec ces anciennes versions de `signal()` pour ce qui concerne le blocage des signaux. Lorsqu'un signal est capturé et que le processus exécute le gestionnaire installé, le noyau ne bloque pas une éventuelle occurrence du même signal. Le gestionnaire peut alors se trouver rappelé au cours de sa propre exécution. Nous allons le démontrer avec ce petit exemple, dans lequel un processus fils envoie deux signaux à court intervalle à son père, lequel utilise un gestionnaire lent, qui compte jusqu'à 3.

exemple_signal_3.c :

```
    #define _XOPEN_SOURCE
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>

    void
gestionnaire (int numero_signal)
{
    int i;
    signal(numero_signal, gestionnaire);
```

```
    fprintf(stdout, "début du gestionnaire de signaux %d \n",
                      numero_signal);
    for (i = 1; i < 4; i ++) {
        fprintf(stdout, "%d\n", i);
        sleep(1);
    }
    fprintf(stdout, "fin du gestionnaire de signaux %d\n", numero_signal);
}

    int
main (void)
{
    signal(SIGUSR1, gestionnaire);
    if (fork() == 0) {
        kill(getppid(), SIGUSR1);
        sleep(1);
        kill(getppid(), SIGUSR1);
    } else {
        while (1)
            pause();
    }
    return EXIT_SUCCESS;
}
```

Voici ce que donne l'exécution de ce programme :

```
$ ./exemple_signal_3
début du gestionnaire de signaux 10
1
début du gestionnaire de signaux 10
1
2
3
fin du gestionnaire de signaux 10
2
3
fin du gestionnaire de signaux 10
  (Contrôle-C)
$
```

Les deux comptages sont enchevêtrés, ce qui n'est pas grave car la variable `i` est allouée de manière automatique dans la pile, et il y a donc deux compteurs différents pour les deux invocations du gestionnaire. Mais cela pourrait se passer autrement si la variable de comptage était statique ou globale. Il suffit de déplacer le `int i` pour le placer en variable globale avant le gestionnaire, et on obtient l'exécution suivante :

```
$ ./exemple_signal_3
début du gestionnaire de signaux 10
1
début du gestionnaire de signaux 10
1
2
3
```

```
fin du gestionnaire de signaux 10
fin du gestionnaire de signaux 10
  (Contrôle-C)
$
```

Cette fois-ci, le compteur global était déjà arrivé à 4 lorsqu'on est revenu dans le premier gestionnaire, celui qui avait lui-même été interrompu par le signal. Pour éviter ce genre de désagrément, la version moderne de `signal()`, disponible sous Linux, bloque automatiquement un signal lorsqu'on exécute son gestionnaire, puis le débloque au retour. On peut le vérifier en supprimant la ligne `#define _XOPEN_SOURCE` et on obtient (même en laissant le compteur en variable globale) :

```
$ ./exemple_signal_3
début du gestionnaire de signaux 10
1
2
3
fin du gestionnaire de signaux 10
début du gestionnaire de signaux 10
1
2
3
fin du gestionnaire de signaux 10
  (Contrôle-C)
$
```

Comme on pouvait s'y attendre, les deux exécutions du gestionnaire de signaux sont séparées. On peut noter au passage que si on rajoute un troisième

```
    sleep(1);
    kill(getppid(), SIGUSR1);
```

dans le processus fils, il n'y a pas de différence d'exécution. Seules deux exécutions du gestionnaire ont lieu. C'est dû au fait que, sous Linux, les signaux classiques ne sont pas empilés, et l'arrivée du troisième `SIGUSR1` se fait alors que le premier gestionnaire n'est pas terminé. Aussi, un seul signal est mis en attente. Remarquons également que lorsqu'on élimine la définition `_XOPEN_SOURCE`, on peut supprimer l'appel `signal()` à l'intérieur du gestionnaire : celui-ci est automatiquement réinstallé, comme on l'a déjà indiqué.

Bien sûr, toutes ces expérimentations tablent sur le fait que l'exécution des processus se fait de manière homogène, sur un système peu chargé. Si tel n'est pas le cas, les temps de commutation entre les processus père et fils, ainsi que les délais de délivrance des signaux, peuvent modifier les comportements de ces exemples.

Nous voyons que la version de `signal()` disponible sous Linux, héritée de celle de BSD, est assez performante et fiable puisqu'elle permet, d'une part, une réinstallation automatique du gestionnaire lorsqu'il est invoqué et, d'autre part, un blocage du signal concerné au sein de son propre gestionnaire. Une dernière question se pose, qui concerne le redémarrage automatique des appels système lents interrompus.

Pour cela, la bibliothèque GlibC nous offre une fonction de contrôle nommée **`siginterrupt()`**.

```
int siginterrupt (int numero, int interrompre);
```

Elle prend en argument un numéro de signal, suivi d'un indicateur booléen. Elle doit être appelée après l'installation du gestionnaire et, si l'indicateur est nul, les appels système lents seront relancés automatiquement. Si l'indicateur est non nul, les appels système échouent, avec une erreur `EINTR` dans `errno`.

Voici un petit programme qui prend une valeur numérique en argument et la transmet à `siginterrupt()` après avoir installé un gestionnaire pour le signal `TSTP` (touche Contrôle-Z). Il exécute ensuite une lecture bloquante depuis le descripteur de fichier 0 (entrée standard). Le programme nous indique à chaque frappe sur Contrôle-Z si la lecture est interrompue ou non. On peut terminer le processus avec Contrôle-C.

exemple_siginterrupt.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    #include <signal.h>


    void
gestionnaire (int numero_signal)
{
    fprintf(stdout, "\n gestionnaire de signaux %d\n", numero_signal);
}

    int
main (int argc, char * argv [])
{
    int i;

    if ((argc != 2) || (sscanf(argv[1], "%d", & i) != 1)) {
        fprintf(stderr, "Syntaxe : %s {0|1}\n", argv [0]);
        exit (EXIT_FAILURE);
    }
    signal(SIGTSTP, gestionnaire);
    siginterrupt(SIGTSTP, i);

    while (1) {
        fprintf(stdout, "appel read()\n");
        if (read(0, &i, sizeof (int)) < 0)
            if (errno == EINTR)
                fprintf(stdout, "EINTR \n");
    }
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution :

```
$ ./exemple_siginterrupt 0
appel read()
  (Contrôle-Z)
```

```
gestionnaire de signaux 20
  (Contrôle-Z)
gestionnaire de signaux 20
  (Contrôle-C)
$ ./exemple_siginterrupt 1
appel read()
  (Contrôle-Z)
gestionnaire de signaux 20
EINTR
appel read()
  (Contrôle-Z)
gestionnaire de signaux 20
EINTR
appel read()
  (Contrôle-C)
$
```

En supprimant la ligne `siginterrupt()`, on peut s'apercevoir que le comportement est identique à « `exemple_siginterrupt 0` ». Les appels système lents sont donc relancés automatiquement sous Linux par défaut. Si, par contre, nous définissons la constante `_XOPEN_SOURCE` comme nous l'avons fait précédemment, en supprimant la ligne `siginterrupt()`, on observe que les appels lents ne sont plus relancés.

Conclusion

Nous voyons donc que l'appel système `signal()` donne accès, sous Linux, avec la bibliothèque GlibC, à des signaux fiables. Nous pouvons aussi compiler des sources datant d'anciens systèmes Unix et se fondant sur un comportement moins fiable de `signal()`, simplement en définissant des constantes symboliques à la compilation (consulter à ce sujet le fichier `/usr/include/features.h`).

Malheureusement, ce n'est pas le cas sur tous les systèmes, aussi est-il préférable d'employer la fonction `sigaction()`, que nous allons étudier dans le prochain chapitre et qui permet un paramétrage plus souple du comportement du programme.

L'appel `signal()` doit surtout être réservé aux cas où l'on désire ignorer un signal ou lui rendre son comportement par défaut. Pour installer un gestionnaire dans une application portable, on préférera `sigaction()`.

7

Gestion portable des signaux

La gestion des signaux de manière portable n'est pas beaucoup plus compliquée que ce que nous avons vu dans le chapitre précédent. L'appel-système `sigaction()` que nous allons étudier tout d'abord permet de réaliser toutes les opérations de configuration du gestionnaire et du comportement des signaux.

Nous examinerons ensuite le principe des ensembles de signaux, qui permettent d'assurer les blocages temporaires avec `sigprocmask()`. Enfin nous observerons les « bonnes manières » d'écrire un gestionnaire de signaux, ce que nous mettrons en pratique avec une étude de l'alarme `SIGALRM`.

Réception des signaux avec l'appel-système sigaction()

La routine **`sigaction()`** prend trois arguments et renvoie un entier valant 0 si elle réussit, et –1 si elle échoue. Le premier argument est le numéro du signal (comme toujours, il faut utiliser la constante symbolique définissant le nom du signal). Les deux autres arguments sont des pointeurs sur des structures `sigaction` (pas d'inquiétude, il n'y a pas d'ambiguïté avec le nom de la routine dans la table des symboles du compilateur). Ces structures définissent précisément le comportement à adopter en cas de réception du signal considéré. Le premier pointeur est le nouveau comportement à programmer, alors que le second pointeur sert à sauvegarder l'ancienne action. Le prototype est donc le suivant :

```
int sigaction (int numero,
               const struct sigaction * nouvelle,
               struct sigaction * ancienne);
```

Si le numéro indiqué est inférieur ou égal à 0, supérieur ou égal à `NSIG`, ou égal à `SIGKILL` ou `SIGSTOP`, `sigaction()` échoue en plaçant `EINVAL` dans `errno`.

Si le pointeur sur la nouvelle structure `sigaction` est `NULL`, aucune modification n'a lieu, seul l'ancien comportement est sauvegardé dans le second pointeur. Parallèlement, si le pointeur sur l'ancienne structure est `NULL`, aucune sauvegarde n'a lieu.

Voyons maintenant le détail de la structure `sigaction`. Celle-ci est définie dans `<sigaction.h>` qui est inclus par `<signal.h>` :

Nom	Type
`sa_handler`	`sighandler_t`
`sa_mask`	`sigset_t`
`sa_flags`	`int`
`sa_restorer`	`void (*) (void)`

Attention

L'ordre des membres de cette structure n'est pas fixé suivant les systèmes, et il a changé entre différentes versions du noyau Linux. Il ne faut donc surtout pas le considérer comme immuable et éviter par exemple d'initialiser la structure de manière statique.

Le premier membre de cette structure correspond à un pointeur sur le gestionnaire du signal, comme nous en transmettions à l'appel-système `signal()`. Le champ `sa_handler` peut également prendre comme valeur `SIG_IGN` pour ignorer le signal ou `SIG_DFL` pour appliquer l'action par défaut. Un gestionnaire doit donc être défini ainsi :

```
void gestionnaire_signal (int numero);
```

Le second membre est du type `sigset_t`, c'est-à-dire un ensemble de signaux. Nous verrons plus bas des fonctions permettant de configurer ce type de données. Cet élément correspond à la liste des signaux qui sont bloqués pendant l'exécution du gestionnaire. Le signal ayant déclenché l'exécution du gestionnaire est automatiquement bloqué, sauf si on demande explicitement le contraire (voir ci-dessous `SA_NODEFER`). Une tentative de blocage de `SIGKILL` ou `SIGCONT` est silencieusement ignorée.

Enfin, le troisième membre `sa_flags` contient un OU binaire entre différentes constantes permettant de configurer le comportement du gestionnaire de signaux :

Nom	Signification
`SA_NOCLDSTOP`	Cette constante ne concerne que le gestionnaire pour le signal `SIGCHLD`. Lorsqu'elle est présente, ce gestionnaire n'est pas invoqué lorsqu'un processus fils a été stoppé temporairement (avec le signal `SIGSTOP` ou `SIGTSTP`). Par contre, il sera appelé pour les processus fils qui se terminent définitivement. Pour tout autre signal que `SIGCHLD`, cette constante est ignorée.
`SA_RESTART`	Lorsqu'elle est présente, cette constante indique que les appels système lents interrompus par le signal concerné sont automatiquement redémarrés. On l'utilise généralement pour tous les signaux, sauf pour `SIGALRM` s'il sert à installer un délai maximal pour une fonction pouvant rester bloquée.
`SA_INTERRUPT`	Cette constante n'est pas définie par SUSv3. Elle a le comportement exactement inverse de `SA_RESTART`, en empêchant le redémarrage automatique des appels système interrompus. Il suffit, pour porter sous Linux des programmes utilisant cette valeur, de la supprimer (éventuellement avec un `#define SA_INTERRUPT 0`) et de ne pas utiliser la constante `SA_RESTART` dans ce cas.

Nom	Signification
SA_NODEFER	Demande explicitement qu'un signal ne soit pas bloqué à l'intérieur de son propre gestionnaire. En réalité, sous Linux, elle est identique à SA_NOMASK, constante qui empêche de bloquer les signaux mentionnés dans sa_mask. Sa portée est donc plus grande. Si on désire obtenir le comportement « classique » de SA_NODEFER, il vaut mieux supprimer le signal considéré de l'ensemble de signaux de sa_mask.
SA_RESETHAND	Lorsqu'un gestionnaire de signaux est invoqué et que cette constante a été fournie à sigaction(), le comportement par défaut est réinstallé pour le signal concerné. C'était le comportement normal des premières versions de signal(), ce que nous avions forcé avec la constante _XOPEN_SOURCE dans notre programme exemple_signal_2.c.
SA_ONESHOT	Cette constante n'est pas définie par SUSv3. Elle est équivalente à SA_RESETHAND.
SA_SIGINFO	Il s'agit d'une valeur décrite par SUSv3 pour les signaux temps-réel, mais qui peut être utilisée aussi pour les signaux classiques. Un gestionnaire de signaux installé avec cette option recevra des informations supplémentaires, en plus du numéro du signal qui l'a déclenché. Le gestionnaire doit accepter trois arguments : le premier est toujours le numéro du signal, le second est un pointeur sur une structure de type siginfo_t, le troisième argument, de type void *, n'est pas documenté dans les sources du noyau. Nous détaillerons cette possibilité plus bas.
SA_ONSTACK	Dans ce cas, le gestionnaire du signal en question utilise une pile différente de celle du reste du programme. Nous fournirons plus loin un exemple d'utilisation de cette possibilité.

Lorsqu'on utilise l'attribut SA_SIGINFO, on considère que la structure sigaction contient un champ supplémentaire, nommé sa_sigaction, permettant de stocker le pointeur sur le gestionnaire. En réalité, tout est souvent implémenté sous forme d'union, une seule et même zone servant à stocker l'adresse des différents types de gestionnaires, mais les prototypes différents permettant une vérification à la compilation. Cela signifie aussi que les noms des membres présents dans la structure sigaction peuvent être en réalité des macros permettant d'accéder à des champs dont le nom est plus complexe, et qu'il faut éviter, sous peine de voir le compilateur échouer, d'appeler une variable sa_sigaction, par exemple.

Le gestionnaire de signaux devra dans ce cas avoir la forme suivante :

```
void gestionnaire_signal (int numero,
                          struct siginfo * info,
                          void * inutile);
```

La structure siginfo peut également être implémentée de manière assez complexe, avec des champs en union. De manière portable, on peut accéder aux membres suivants, définis par SUSv3 :

Nom	Signification
si_signo	Indique le numéro de signal.
si_sigval	Est à son tour une union qui n'est utilisée qu'avec les signaux temps-réel, et que nous détaillerons donc ultérieurement.
si_code	Indique la provenance du signal. Il s'agit d'une combinaison binaire par OU entre diverses constantes, variant en fonction du signal reçu. Si si_code est strictement positif, le signal provient du noyau. Sinon, il provient d'un utilisateur (même root). Nous verrons des exemples avec les signaux temps-réel.

Nom	Signification
si_errno	Ce champ contient la valeur de errno lors de l'invocation. Permet par exemple de la rétablir en sortie de gestionnaire.
si_pid et si_uid	Ces membres ne sont valides que si le signal provient d'un utilisateur (si_code négatif ou nul), ou si le signal SIGCHLD a été émis par le noyau. Ils identifient l'émetteur du signal ou le processus fils qui s'est terminé.

Les informations fournies grâce à la structure siginfo peuvent être très importantes en termes de sécurité pour des programmes susceptibles d'avoir des privilèges particuliers (Set-UID *root*). Cela permet de vérifier que le signal est bien émis par le noyau et non par un utilisateur essayant d'exploiter une faille de sécurité. Nous reparlerons de ces données dans le prochain chapitre, car elles concernent également les signaux temps-réel.

Lorsqu'on utilise l'attribut SA_ONSTACK lors de l'invocation de sigaction(), la pile est alors sauvegardée. Il faut déclarer une pile différente à l'aide de l'appel système **sigaltstack()**. Les variables locales automatiques étant utilisées dans les routines, y compris les gestionnaires de signaux alloués dans la pile, il peut être intéressant dans certains cas de réserver avec malloc() une place mémoire suffisamment importante pour accueillir des variables assez volumineuses. Les constantes **MINSIGSTKSZ** et **SIGSTKSZ**, définies dans <signal.h>, correspondent respectivement à la taille minimale et à la taille optimale pour la pile réservée à un gestionnaire de signaux. Une structure sigaltstack, définie dans <signal.h>, contient trois champs :

Nom	Type	Signification
ss_sp	void *	Pointeur sur la pile
ss_flags	int	Attributs
ss_size	size_t	Taille de la pile

On alloue donc la place voulue dans le champ ss_sp d'une variable stack_t, puis on invoque l'appel sigaltstack() en lui fournissant cette nouvelle pile. Elle sera alors utilisée pour tous les gestionnaires de signaux qui emploient l'attribut SA_ONSTACK dans leur installation par sigaction(). Le prototype de sigaltstack() est le suivant :

```
int sigaltstack (stack_t * nouvelle, stack_t * ancienne);
```

L'appel permet éventuellement de sauvegarder l'ancienne pile en utilisant un second argument non nul. Dans le cas où le premier argument est NULL, aucune modification n'a lieu, on obtient simplement l'état actuel de la pile. Cela permet de vérifier si cette pile est en cours d'utilisation ou non. Les deux constantes symboliques SS_DISABLE et SS_ONSTACK indiquent respectivement dans le champ ss_flags que la pile est désactivée ou qu'elle est en cours d'utilisation.

Nous verrons un exemple d'utilisation de pile spécifique pour les gestionnaires de signaux à la fin du paragraphe sur les exemples d'utilisation de sigaction().

Configuration des ensembles de signaux

Avant de voir des exemples d'utilisation de `sigaction()`, nous allons regarder les différentes primitives permettant de modifier les ensembles de signaux de type `sigset_t`. Ce type est opaque, et il faut absolument utiliser les routines décrites ci-dessous pour y accéder.

À titre d'exemple, nous rappelons que le noyau Linux 2.0 définissait `sigset_t` comme un `unsigned long`, le noyau 2.2 comme un tableau de 2 `unsigned long`, et la bibliothèque GlibC comme un tableau de 32 `unsigned long` (se réservant de la place pour des extensions jusqu'à 1 024 signaux). La définition réelle du type `sigset_t` peut donc varier suivant les machines, mais également selon les versions du noyau et même le type de fichier d'en-tête utilisé (noyau ou bibliothèque C).

Les routines suivantes sont définies par SUSv3 :

```
int sigemptyset  (sigset_t * ensemble);
int sigfillset   (sigset_t * ensemble);
int sigaddset    (sigset_t * ensemble, int numero_signal);
int sigdelset    (sigset_t * ensemble, int numero_signal);
int sigismember  (const sigset_t * ensemble, int numero_signal);
```

La première routine, **`sigemptyset()`**, permet de vider un ensemble, c'est-à-dire de l'initialiser sans aucun signal. Il ne faut **pas** utiliser une initialisation du genre `ensemble=0`, car elle n'est pas suffisante dans le cas où le type `sigset_t` est un tableau (dans la GlibC, par exemple). Parallèlement, **`sigfillset()`** permet de remplir un ensemble avec tous les signaux connus sur le système. Ces deux routines renvoient 0 si elles réussissent et –1 sinon, c'est-à-dire si `ensemble` vaut `NULL` ou pointe sur une zone mémoire invalide.

Les routines **`sigaddset()`** et **`sigdelset()`** permettent respectivement d'ajouter un signal à un ensemble ou d'en supprimer. Elles renvoient 0 si elles réussissent ou –1 si elles échouent (si le numéro de signal est invalide ou si `ensemble` est `NULL`). Le fait d'ajouter un signal à un ensemble qui le contient déjà ou de supprimer un signal d'un ensemble auquel il n'appartient pas ne constitue pas une erreur.

La dernière routine, **`sigismember()`**, permet de savoir si un signal appartient à un ensemble ou pas ; elle renvoie 1 si c'est le cas, ou 0 sinon. Elle peut également renvoyer –1 en cas d'erreur.

La bibliothèque GlibC ajoute trois fonctionnalités qui sont des extensions Gnu (nécessitant donc la constante `_GNU_SOURCE` à la compilation). Elles permettent de manipuler les ensembles de signaux de manière globale :

```
int sigisemptyset (const sigset_t * ensemble)
int sigandset (sigset_t * ensemble_resultat,
               const sigset_t * ensemble_1,
               const sigset_t * ensemble_2)
int sigorset  (sigset_t * ensemble_resultat,
               const sigset_t * ensemble_1,
               const sigset_t * ensemble_2)
```

`sigisemptyset()` indique si l'ensemble considéré est vide. **`sigandset()`** permet d'effectuer un ET binaire entre deux ensembles de signaux et d'obtenir dans l'ensemble résultat la liste des signaux qui leur sont communs. **`sigorset()`** permet symétriquement d'effectuer un OU binaire pour obtenir en résultat l'ensemble des signaux présents dans l'un ou l'autre des deux ensembles.

Exemples d'utilisation de sigaction()

Notre premier exemple consistera à installer deux gestionnaires : l'un pour `SIGQUIT` (que nous déclenchons au clavier avec Contrôle-AltGr-\), qui ne fera pas redémarrer les appels système lents interrompus, le second, celui de `SIGINT` (Contrôle-C), aura pour particularité de ne pas se réinstaller automatiquement. La seconde occurrence de `SIGINT` déclenchera donc le comportement par défaut et arrêtera le processus.

exemple_sigaction_1.c :

```
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <errno.h>

    void
gestionnaire (int numero)
{
    switch (numero) {
        case SIGQUIT :
            fprintf(stdout, "\n SIGQUIT reçu \n"); fflush(stdout);
            break;
        case SIGINT :
            fprintf(stdout, "\n SIGINT reçu \n"); fflush(stdout);
            break;
    }
}

    int
main (void)
{
    struct sigaction action;

    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = 0;
    if (sigaction(SIGQUIT, & action, NULL) != 0) {
        fprintf(stderr, "Erreur %d \n", errno);
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = SA_RESTART | SA_RESETHAND;
    if (sigaction(SIGINT, & action, NULL) != 0) {
        fprintf(stderr, "Erreur %d \n", errno);
        exit(EXIT_FAILURE);
    }
    /* Lecture continue, pour avoir un appel système lent bloqué */
    while (1) {
        int i;
        fprintf(stdout, "appel read()\n");
        if (read(0, &i, sizeof (int)) < 0)
```

```
            if (errno == EINTR)
                fprintf(stdout, "EINTR \n");
    }
    return EXIT_SUCCESS;
}
```

L'exécution de ce programme montre bien les différences de caractéristiques entre les signaux :

```
$ ./exemple_sigaction_1
appel read()
                        Ctrl-AltGr-\
SIGQUIT reçu
EINTR
appel read()
                        Ctrl-C
SIGINT reçu
                        Ctrl-AltGr-\
SIGQUIT reçu
EINTR
appel read()
                        Ctrl-C
$
```

`SIGQUIT` interrompt bien l'appel `read()`, mais pas `SIGINT`. De même, le gestionnaire de `SIGQUIT` reste installé et peut être appelé une seconde fois, alors que `SIGINT` reprend son comportement par défaut et termine le processus la seconde fois.

Nous n'avons pas sauvegardé l'ancien gestionnaire de signaux lors de l'appel de `sigaction()` (troisième argument). Il est pourtant nécessaire de le faire si nous installons temporairement une gestion de signaux propre à une seule partie du programme. De même, lorsqu'un processus est lancé en arrière-plan par un shell ne gérant pas le contrôle des jobs, celui-ci force certains signaux à être ignorés. Les signaux concernés sont `SIGINT` et `SIGQUIT`. Dans le cas d'un shell sans contrôle de jobs, ces signaux seraient transmis autant au processus en arrière-plan qu'à celui en avant-plan.

Voici un exemple de programme permettant d'afficher les signaux dont le comportement n'est pas celui par défaut.

exemple_sigaction_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <errno.h>

    int
main (void)
{
    int i;
    struct sigaction action;
    for (i = 1; i < NSIG; i ++) {
        if (sigaction(i, NULL, & action) != 0)
            fprintf(stderr, "Erreur sigaction %d \n", errno);
        if (action.sa_handler != SIG_DFL) {
```

```
            fprintf(stdout, "%d (%s) comportement ",
                    i, sys_siglist [i]);
                if (action.sa_handler == SIG_IGN)
                    fprintf(stdout, ": ignorer \n");
                else
                    fprintf(stdout, "personnalisé \n");
        }
    }
    return EXIT_SUCCESS;
}
```

Pour l'exécuter de manière probante, il faut arrêter le contrôle des jobs. Sous `bash`, cela s'effectue à l'aide de la commande `set +m`. Au début de notre exemple, `bash` a un contrôle des jobs activé.

```
$ ./exemple_sigaction_2
$ ./exemple_sigaction_2 &
[1] 983
[1]+  Done                ./exemple_sigaction_2
$ set +m
$ ./exemple_sigaction_2
$ ./exemple_sigaction_2 &
[1] 985
2 (Interrupt) comportement : ignorer
$ 3 (Quit) comportement : ignorer
$
```

On voit qu'il n'y a effectivement de différence que pour les processus en arrière-plan si le contrôle des jobs est désactivé. Il vaut donc mieux vérifier, au moment de l'installation d'un gestionnaire pour ces signaux, si le shell ne les a pas volontairement ignorés. Dans ce cas, on les laisse inchangés :

```
struct sigaction action, ancienne;
action.sa_handler = gestionnaire;
/* ... initialisation de action ... */
if (sigaction(SIGQUIT, & action, & ancienne) != 0)
    /* ... gestion d'erreur ... */
if (ancienne.sa_handler != SIG_DFL) {
    /* réinstallation du comportement original */
    sigaction(SIGQUIT, & ancienne, NULL);
}
```

Ceci n'est important que pour `SIGQUIT` et `SIGINT`.

Profitons de cet exemple pour préciser le comportement des signaux face aux appels système `fork()` et `exec()` utilisés notamment par le shell. Lors d'un `fork()`, le processus fils reçoit le même masque de blocage des signaux que son père. Les actions des signaux sont également les mêmes, y compris les gestionnaires installés par le programme. Par contre, les signaux en attente n'appartiennent qu'au processus père.

Lors d'un `exec()`, le masque des signaux bloqués est conservé. Les signaux ignorés le restent. C'est comme cela que le shell nous transmet le comportement décrit ci-dessus pour `SIGINT` et `SIGQUIT`. Mais les signaux qui étaient capturés par un gestionnaire reprennent leur comportement par défaut. C'est logique car l'ancienne adresse du gestionnaire de signaux n'a plus aucune signification dans le nouvel espace de code du programme exécuté.

On peut s'interroger sur la pertinence de mélanger dans un même programme les appels à `signal()` et à `sigaction()`. Cela ne pose aucun problème majeur sous Linux avec GlibC. Le seul inconvénient vient du fait que `signal()` ne peut pas sauvegarder et rétablir ultérieurement autant d'informations sur le gestionnaire de signaux que `sigaction()`. Ce dernier en effet peut sauver et réinstaller les attributs comme `NOCLDSTOP` ou `NODEFER`, au contraire de `signal()`.

Lorsqu'il faut sauvegarder un comportement pour le restituer plus tard, il faut donc impérativement utiliser `sigaction()`, sauf si tout le programme n'utilise que `signal()`.

Lorsqu'on installe un gestionnaire avec `signal()` sous Linux et qu'on examine le comportement du signal avec `sigaction()`, on retrouve dans le champ `sa_handler` la même adresse que celle de la routine installée avec `signal()`. Ceci n'est toutefois pas du tout généralisable à d'autres systèmes, et il ne faut pas s'appuyer sur ce comportement. Voici un exemple de test.

exemple_sigaction_3.c :

```
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>

    void
gestionnaire (int inutilise)
{ }

    int
main (void)
{
    struct sigaction action;

    signal(SIGUSR1, gestionnaire);

    sigaction(SIGUSR1, NULL, & action);
    if (action.sa_handler == gestionnaire)
        fprintf(stdout, "Même adresse \n");
    else
        fprintf(stdout, "Adresse différente \n");
    return EXIT_SUCCESS;
}
```

Sous Linux, pas de surprise :

```
$ uname -sr
Linux 2.0.31
$ ./exemple_sigaction_3
Même adresse
$
```

```
$ uname -sr
Linux 2.6.9
$ ./exemple_sigaction_3
Même adresse
$
```

Nous terminerons cette section avec un exemple d'installation d'une pile spécifique pour les gestionnaires de signaux. Nous allons mettre en place un gestionnaire commun pour les signaux `SIGQUIT` et `SIGTERM`, qui vérifiera si la pile est en cours d'utilisation ou non en examinant le champ `ss_flags`. Nous n'installerons la pile spéciale que pour le signal `SIGQUIT`, ce qui nous permettra de vérifier la différence entre les deux signaux.

exemple_sigaltstack.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <errno.h>

    void
gestionnaire (int numero_signal)
{
    stack_t pile;

    fprintf(stdout, "\n Signal %d reçu \n", numero_signal);
    if (sigaltstack(NULL, & pile) != 0) {
        fprintf(stderr, "Erreur sigaltstack %d \n", errno);
        return;
    }
    if (pile.ss_flags & SS_DISABLE)
        fprintf(stdout, "La pile spéciale est inactive \n");
    else
        fprintf(stdout, "La pile spéciale est active \n");
    if (pile.ss_flags & SS_ONSTACK)
        fprintf(stdout, "Pile spéciale en cours d'utilisation \n");
    else
        fprintf(stdout, "Pile spéciale pas utilisée actuellement \n");
}

    int
main (void)
{
    stack_t pile;
    struct sigaction action;

    if ((pile.ss_sp = malloc(SIGSTKSZ)) == NULL) {
        fprintf(stderr, "Pas assez de mémoire \n");
        exit(EXIT_FAILURE);
    }
    pile.ss_size = SIGSTKSZ;
    pile.ss_flags = 0;
    if (sigaltstack(& pile, NULL) != 0) {
        fprintf(stderr, "Erreur sigaltstack() %d \n", errno);
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
```

```
    action.sa_flags = SA_RESTART | SA_ONSTACK;
    if (sigaction(SIGQUIT, & action, NULL) != 0) {
        fprintf(stderr, "Erreur sigaction() %d \n", errno);
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire;
    sigemptyset(& (action . sa_mask));
    action.sa_flags = SA_RESTART;
    if (sigaction(SIGTERM, & action, NULL) != 0) {
        fprintf(stderr, "Erreur sigaction() %d \n", errno);
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "PID = %ld \n", (long) getpid());
    fflush(stdout);
    while (1)
        pause();
    return EXIT_SUCCESS;
}
```

Voici un exemple d'utilisation :

```
$ ./exemple_sigaltstack
PID = 815
                    $ kill -QUIT 815
Signal 3 reçu
La pile spéciale est active
Pile spéciale en cours d'utilisation
                    $ kill -TERM 815
Signal 15 reçu
La pile spéciale est active
Pile spéciale pas utilisée actuellement
                    $ kill -INT 815
$
```

Comme prévu, la pile spéciale des signaux reste active en permanence, mais elle n'est utilisée que lorsque le gestionnaire est invoqué par `SIGQUIT`.

Blocage des signaux

Nous avons mentionné à plusieurs reprises qu'un processus pouvait bloquer à volonté un ensemble de signaux, sauf `SIGKILL` et `SIGSTOP`. Cette opération se fait principalement grâce à l'appel système **`sigprocmask()`**. Cette routine est très complète puisqu'elle permet aussi bien de :

- bloquer ou débloquer des signaux ;
- fixer un nouveau masque complet ;
- consulter l'ancien masque de blocage.

Le prototype de `sigprocmask()` est le suivant :

```
int sigprocmask (int methode,
                 const sigset_t * ensemble,
                 sigset_t * ancien);
```

Quelle que soit la méthode choisie, si le troisième argument `ancien` est un pointeur non `NULL`, il sera rempli avec le masque actuel de blocage des signaux. Le premier argument permet d'indiquer l'action attendue, par l'intermédiaire de l'une des constantes symboliques suivantes :

Nom	Signification
`SIG_BLOCK`	On ajoute la liste des signaux contenus dans l'ensemble transmis en second argument au masque de blocage des signaux. Il s'agit d'une addition au masque en cours.
`SIG_UNBLOCK`	On retire les signaux contenus dans l'ensemble en second argument au masque de blocage des signaux. S'il existe un ou plusieurs signaux débloqués en attente, au moins un de ces signaux est immédiatement délivré au processus, avant le retour de l'appel système `sigprocmask()`.
`SIG_SETMASK`	Le second argument est utilisé directement comme masque de blocage pour les signaux. Comme pour `SIG_UNBLOCK`, la modification du masque peut entraîner le déblocage d'un ou de plusieurs signaux en attente. L'un au moins de ces signaux est alors délivré immédiatement avant le retour de `sigprocmask()`.

La fonction `sigprocmask()` renvoie 0 si elle réussit, et –1 en cas d'erreur, c'est-à-dire avec `errno` valant `EINVAL` en cas de méthode inexistante ou `EFAULT` si l'un des pointeurs est mal initialisé. Les signaux `SIGKILL` et `SIGSTOP` sont silencieusement éliminés du masque transmis en second argument, sans déclencher d'erreur. Il est donc possible de transmettre un ensemble de signaux remplis avec `sigfillset()` pour tout bloquer ou tout débloquer, sans se soucier de `SIGKILL` et `SIGSTOP`.

L'appel système `sigprocmask()` doit remplacer totalement les anciens appels **`sigblock()`**, **`siggetmask()`**, **`sigsetmask()`**, **`sigmask()`** qui sont désormais obsolètes.

L'utilité principale d'un blocage des signaux est la protection des portions critiques de code. Imaginons qu'un gestionnaire modifie une variable globale comme une structure. Lorsque le programme principal est en train de lire ou de modifier cette variable, il risque d'être interrompu par ce signal au milieu de la lecture et d'avoir des incohérences entre les premiers et les derniers champs lus. L'effet peut être encore pire avec une chaîne de caractères. Pour éviter cette situation, on bloque temporairement l'arrivée du signal concerné pendant la lecture ou la modification de la variable globale.

Imaginons qu'on ait une variable globale du type :

```
typedef struct {
    double X;
    double Y;
}  point_t;

point_t centre, pointeur;
```

Que le gestionnaire de SIGUSR1 modifie la variable `centre` :

```
void gestionnaire_sigusr1 (int inutilise)
{
    centre.X = pointeur.X;
    centre.Y = pointeur.Y;
 }
```

On protégera dans le corps du programme l'accès à cette variable :

```
sigset_t ensemble, ancien;

sigemptyset(& ensemble);
sigaddset(& ensemble, SIGUSR1);

sigprocmask(SIG_BLOCK, & ensemble, & ancien);
X1 = centre.X * zoom;        /* Voici la portion critique */
Y1 = centre.Y * zoom;        /* protégée de SIGUSR1       */
sigprocmask(SIG_SETMASK, & ancien, NULL);

cercle(centre.X, centre.Y, rayon);
```

Ceci peut paraître un peu lourd, mais l'ensemble en question n'a besoin d'être initialisé qu'une seule fois, et on peut aisément définir des macros pour encadrer les portions de code critiques.

Il est important qu'un processus puisse consulter la liste des signaux bloqués en attente, sans pour autant en demander la délivrance immédiate. Cela s'effectue à l'aide de l'appel système **`sigpending()`**, dont le prototype est :

```
int sigpending (sigset_t * ensemble);
```

Comme on pouvait s'y attendre, cette routine remplit l'ensemble transmis en argument avec les signaux en attente. Voici un programme qui permet d'en voir le fonctionnement. Tout d'abord, nous installons un gestionnaire qui indique le numéro du signal reçu, et ce pour tous les signaux. Ce gestionnaire ne relance pas les appels système lents interrompus. Ensuite, nous bloquons tous les signaux, sauf `SIGINT` (Contrôle-C), et nous lançons une lecture bloquée, pendant laquelle nous pouvons appuyer sur des touches spéciales du clavier (Contrôle-Z, Contrôle-AltGr-\) ou envoyer des signaux depuis une autre console. Lorsque nous appuyons sur Contrôle-C, `SIGINT` non bloqué fait échouer la lecture, et le programme continue, nous indiquant la liste des signaux bloqués en attente. Nous débloquons alors tous les signaux et nous les regardons arriver.

exemple_sigpending.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <errno.h>
    #include <unistd.h>

#ifdef _POSIX_REALTIME_SIGNALS
    #define NB_SIG_CLASSIQUES       SIGRTMIN
#else
    #define NB_SIG_CLASSIQUES       NSIG
#endif

    void
gestionnaire (int numero_signal)
{
    fprintf(stdout, "%d (%s) reçu \n", numero_signal,
                        sys_siglist[numero_signal]);
}
```

```
    int
main (void)
{
    int i;
    struct sigaction action;
    sigset_t ensemble;

    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = 0; /* Pas de SA_RESTART */
    for (i = 1; i < NSIG; i ++)
        if (sigaction(i, & action, NULL) != 0)
                fprintf(stderr, "%ld : %d pas capturé \n",
                                (long) getpid(), i);
    /* On bloque tout sauf SIGINT */
    sigfillset(& ensemble);
    sigdelset(& ensemble, SIGINT);
    sigprocmask(SIG_BLOCK, & ensemble, NULL);

    /* un appel système lent bloqué */
    read(0, & i, sizeof(int));

    /* Voyons maintenant qui est en attente */
    sigpending(& ensemble);
    for (i = 1; i < NB_SIG_CLASSIQUES; i ++)
        if (sigismember(& ensemble, i))
            fprintf(stdout, "en attente %d (%s)\n",
                    i, sys_siglist[i]);
    /* On débloque tous les signaux pour les voir arriver */
    sigemptyset(& ensemble);
    sigprocmask(SIG_SETMASK, & ensemble, NULL);
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution avec, en seconde colonne, les commandes saisies sur une autre console.

```
$ ./exemple_sigpending
4419 : 9 pas capturé
4419 : 19 pas capturé
  (Contrôle-Z)
  (Contrôle-AltGr-\)
                                $ kill -TERM 4419
                                $ kill -USR1 4419
                                $ kill -PIPE 4419
                                $ kill -PIPE 4419
  (Contrôle-C)
2 (Interrupt) reçu
en attente 3 (Quit)
en attente 10 (User defined signal 1)
en attente 13 (Broken pipe)
en attente 15 (Terminated)
en attente 20 (Stopped)
```

```
3 (Quit) reçu
10 (User defined signal 1) reçu
13 (Broken pipe) reçu
15 (Terminated) reçu
20 (Stopped) reçu
$
```

On remarque deux choses : d'abord le signal `PIPE` a été émis deux fois mais n'est reçu qu'en un seul exemplaire. C'est normal, Linux n'empile pas les signaux classiques. Ensuite, les signaux sont délivrés dans l'ordre numérique et non pas dans leur ordre chronologique d'arrivée. C'est en fait une conséquence de la première remarque. Notez bien que la délivrance par ordre numérique croissant est simplement un détail d'implémentation sous Linux, et n'est absolument pas normalisée par SUSv3.

Il n'y a pas non plus de notion de priorité entre les signaux classiques. Si on désire introduire un ordre précis, on peut débloquer signal par signal en utilisant plusieurs fois de suite `sigprocmask()`.

Attente d'un signal

Il y a de nombreuses occasions dans un programme où on désire attendre passivement l'arrivée d'un signal qui se produira de manière totalement asynchrone (par exemple pour se synchroniser avec un processus fils).

Pour cela, l'appel-système le plus évident est **`pause()`**. Celui-ci endort le processus jusqu'à ce qu'il soit interrompu par n'importe quel signal. Il est déclaré dans `<unistd.h>`, ainsi :

```
int pause (void);
```

Cet appel système renvoie toujours –1 en remplissant `errno` avec la valeur `EINTR`.

Le problème qui se pose souvent est d'arriver à encadrer correctement `pause()`, de façon à éviter de perdre des signaux. Imaginons que `SIGUSR1` dispose d'un gestionnaire faisant passer à 0 une variable globale nommée `attente`. On désire bloquer l'exécution du programme jusqu'à ce que cette variable ait changé. La première version naïve de ce programme serait :

```
attente = 1;
while (attente != 0)
    pause();
```

Nous utilisons une boucle `while()` car il se peut que l'appel système `pause()` soit interrompu par un autre signal qui ne modifie pas la variable.

Le gros problème de ce type de comportement est que le signal peut arriver entre le test `(attente!=0)` et l'appel `pause()`. Si le signal modifie la variable à ce moment-là, et si le programme ne reçoit plus d'autres signaux, le processus restera bloqué indéfiniment dans `pause()`.

Un autre problème se pose, car on peut avoir d'autres tâches à accomplir dans la boucle (en rapport, par exemple, avec les autres signaux reçus), et le signal peut éventuellement arriver dans ces périodes gênantes.

Pour éviter cette situation, on pourrait vouloir bloquer le signal temporairement ainsi :

```
sigset_t  ensemble;
sigset_t  ancien;
```

```
sigemptyset(& ensemble);
sigaddset(& ensemble, SIGUSR1);
sigprocmask(SIG_BLOCK, & ensemble, & ancien);
attente = 1;
while (attente != 0) {
    sigprocmask(SIG_UNBLOCK, & ensemble, NULL);
    pause();
    sigprocmask(SIG_BLOCK, & ensemble, NULL);
    /* traitement pour les autres signaux */
}
sigprocmask(SIG_SETMASK, & ancien, NULL);
```

Malheureusement, nous avons indiqué qu'un signal bloqué en attente était délivré avant le retour de `sigprocmask()`, qui le débloque. Nous avons ainsi encore augmenté le risque d'un blocage infini dans `pause()`. On pourrait vérifier de nouveau `(attente!=0)` entre `sigprocmask()` et `pause()`, mais le signal pourrait encore s'infiltrer entre ces deux étapes et bloquer indéfiniment.

Il existe un appel système, **`sigsuspend()`**, qui permet de manière atomique de modifier le masque des signaux et de bloquer en attente. Une fois qu'un signal non bloqué arrive, `sigsuspend()` restitue le masque original avant de se terminer. Son prototype est :

```
int sigsuspend (const sigset_t * ensemble);
```

Attention

L'ensemble transmis est celui des signaux qu'on bloque, pas celui des signaux qu'on attend.

Voici comment l'utiliser, pour attendre l'arrivée de `SIGUSR1`.

```
sigset_t    ensemble;
sigset_t    ancien;
int         sigusr1_dans_masque = 0;

sigemptyset(& ensemble);
sigaddset(& ensemble, SIGUSR1);
sigprocmask(SIG_BLOCK, & ensemble, & ancien);
if (sigismember(& ancien, SIGUSR1)) {
    sigdelset(& ancien, SIGUSR1);
    sigusr1_dans_masque = 1;
}
/* initialisation, etc. */
attente = 1;
while (attente != 0) {
    sigsuspend(& ancien);
    /* traitement pour les éventuels autres signaux */
}
if (sigusr1_dans_masque)
    sigaddset(& ancien, SIGUSR1);
sigprocmask(SIG_SETMASK, & ancien, NULL);
```

On remarquera que nous prenons soin de restituer l'ancien masque de blocage des signaux en sortie de routine, et qu'en transmettant cet ancien masque à `sigsuspend()`, nous permettons l'arrivée d'autres signaux que `SIGUSR1`.

Signalons qu'il existe un appel système **`sigpause()`** obsolète, qui fonctionnait approximativement comme `sigsuspend()`, mais en utilisant un masque de signaux contenu obligatoirement dans un entier de type `int`.

Écriture correcte d'un gestionnaire de signaux

En théorie, suivant le C Ansi, la seule chose qu'on puisse faire dans un gestionnaire de signaux est de modifier une ou plusieurs variables globales de type `sig_atomic_t` (défini dans `<signal.h>`). Il s'agit d'un type entier, souvent un `int` d'ailleurs, que le processeur peut traiter de manière atomique, c'est-à-dire sans risque d'être interrompu par un signal. Il faut déclarer la variable globale avec l'indicateur `volatile` pour signaler au compilateur qu'elle peut être modifiée à tout moment, et pour qu'il ne se livre pas à des optimisations (par exemple en gardant la valeur dans un registre du processeur). Dans ce cas extrême, le gestionnaire ne fait que positionner l'état d'une variable globale, qui est ensuite consultée dans le corps du programme.

Nous avons vu qu'avec une gestion correcte des blocages des signaux, il est en fait possible d'accéder à n'importe quel type de données globales. Le même problème peut toutefois se présenter si un signal non bloqué arrive alors qu'on est déjà dans l'exécution du gestionnaire d'un autre signal. C'est à ce moment que le champ `sa_mask` de la structure `sigaction` prend tout son sens.

Une autre difficulté est de savoir si on peut invoquer, depuis un gestionnaire de signaux, un appel système ou une fonction de bibliothèque. Une grande partie des fonctions de bibliothèque ne sont pas réentrantes. Cela signifie qu'elles utilisent en interne des variables statiques ou des structures de données complexes, comme `malloc()`, et qu'une fonction interrompue en cours de travail dans le corps principal du programme ne doit pas être rappelée depuis un gestionnaire de signaux. Prenons l'exemple de la fonction `ctime()`. Celle-ci prend en argument un pointeur sur une date du type `time_t`, et renvoie un pointeur sur une chaîne de caractères décrivant la date et l'heure. Cette chaîne est allouée de manière statique et est écrasée à chaque appel. Si elle est invoquée dans le corps du programme, interrompue et rappelée dans le gestionnaire de signaux, au retour de ce dernier, la valeur renvoyée dans le corps du programme principal ne sera pas celle qui est attendue. Les fonctions de bibliothèque qui utilisent des variables statiques le mentionnent dans leurs pages de manuel. Il est donc nécessaire de les consulter avant d'introduire la fonction dans un gestionnaire.

Il est important également d'éviter résolument les fonctions qui font appel indirectement à `malloc()` ou à `free()`, comme `tempnam()`.

Il existe une liste minimale, définie par SUSv3, des appels système pouvant être invoqués depuis un gestionnaire de signaux (*async-signal-safe*). Ces appels sont soit réentrants, soit non-interruptibles par les signaux. Remarquons qu'un appel async-signal-safe est également utilisable sans restriction en programmation multithread, mais que la réciproque n'est pas toujours vraie (par exemple `malloc()` est correct pour les programmes multithreads mais ne doit pas être invoqué dans un gestionnaire de signaux).

Les appels « *async-signal-safe* » de SUSv3 sont :

```
_Exit, _exit,
abort, accept, access, aio_error, aio_return, aio_suspend, alarm,
bind,
cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed, chdir, chmod, chown,
➥ clock_gettime, close, connect, creat,
dup, dup2,
execle, execve,
fchmod, fchown, fcntl, fdatasync, fork, fpathconf, fstat, fsync, ftruncate,
getegid, geteuid, getgid, getgroups, getpeername, getpgrp, getpid, getppid,
➥ getsockname, getsockopt, getuid,
kill,
link, listen, lseek,lstat,
mkdir, mkfifo,
open,
pathconf, pause, pipe, poll, posix_trace_event, pselect,
raise, read, readlink, recv, recvfrom, recvmsg, rename, rmdir,
select, sem_post, send, sendmsg, sendto, setgid, setpgid, setsid, setsockopt setuid,
➥ shutdown, sigaction, sigaddset, sigdelset, sigemptyset, sigfillset, sigismember,
➥ sleep, signal, sigpause, sigpending, sigprocmask, sigqueue, sigset, sigsuspend,
➥ socket, socketpair, stat, symlink, sysconf,
tcdrain, tcflow, tcflush, tcgetattr, tcgetpgrp, tcsendbreak, tcsetattr, tcsetpgrp,
➥ time, timer_getoverrun, timer_gettime, timer_settime, times,
umask, uname, unlink, utime,
wait, waitpid, write
```

Les fonctions d'entrée-sortie sur des flux, `fprintf()` par exemple, ne doivent pas être utilisées sur le même flux entre le programme principal et un gestionnaire, à cause du risque important de mélange anarchique des données. Par contre, il est tout à fait possible de réserver un flux de données pour le gestionnaire (`stderr` par exemple), ou de l'employer si nous sommes sûrs que le programme principal ne l'utilise pas au même moment.

Il est très important qu'un gestionnaire de signaux employant le moindre appel système sauvegarde le contenu de la variable globale `errno` en entrée du gestionnaire et qu'il la restitue en sortie. Cette variable est en effet modifiée par la plupart des fonctions système, et le signal peut très bien s'être déclenché au moment où le programme principal terminait un appel système et se préparait à consulter `errno`.

Notons, pour terminer, que dans les programmes s'appuyant sur l'environnement graphique X11, il ne faut en aucun cas utiliser les routines graphiques (Xlib, Xt, Motif…), qui ne sont pas réentrantes. Il faut alors utiliser des variables globales comme indicateurs des actions à exécuter dans le corps même du programme.

Il peut arriver que le travail du gestionnaire soit d'effectuer simplement un peu de nettoyage avant de terminer le processus. L'arrêt peut se faire avec l'appel système `_exit()` ou `exit()`. Néanmoins, il est souvent préférable que le processus père sache que son fils a été tué par un signal et qu'il ne s'est pas terminé normalement. Pour cela, il faut reprogrammer le comportement original du signal et se l'envoyer à nouveau. Bien sûr, cela ne fonctionne qu'avec des signaux qui terminent par défaut le processus (comme `SIGTERM`). De plus, dans certains cas (comme `SIGSEGV`), un fichier d'image mémoire `core` sera créé.

exemple_fatal.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>

    void
gestionnaire_signal_fatal (int numero)
{
    /* Effectuer le nettoyage :                    */
    /*   Couper proprement les connexions réseau    */
    /*   Supprimer les fichiers de verrouillage     */
    /*   Tuer éventuellement les processus fils     */
    fprintf(stdout, "\n Je fais le ménage !\n");
    fflush(stdout);
    signal(numero, SIG_DFL);
    raise(numero);
}

    int
main ()
{
    struct sigaction action;
    action.sa_handler = gestionnaire_signal_fatal;
    action.sa_flags = 0;
    sigfillset(& action.sa_mask);
    fprintf(stdout, "mon pid est %ld\n", (long) getpid ());
    if ((sigaction(SIGTERM, & action, NULL) != 0)
     || (sigaction(SIGSEGV, & action, NULL) != 0)) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
    while (1)
        pause();
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution. On envoie les signaux depuis une autre console. Le shell (`bash` en l'occurrence) nous indique que les processus ont été tués par des signaux.

```
$ ./exemple_fatal
mon pid est 6032
                                    $ kill -TERM 6032
Je fais le ménage !
Terminated
$ ./exemple_fatal
mon pid est 6033
                                    $ kill -SEGV 6033
Je fais le ménage !
Segmentation fault (core dumped)
$
```

Les messages *Terminated* ou *Segmentation fault* sont affichés par le shell lorsqu'il se rend compte que son processus se termine anormalement.

Il y a ici un point de sécurité à noter : certains programmes, souvent Set-UID *root*, disposent temporairement en mémoire de données que même l'utilisateur qui les a lancés ne doit pas connaître. Cela peut concerner par exemple le fichier *shadow* des mots de passe ou les informations d'authentification servant à établir la liaison avec un fournisseur d'accès Internet. Dans ce genre d'application, il est important que le programme écrase ces données sensibles avant de laisser le gestionnaire par défaut créer une éventuelle image mémoire `core` qu'on pourrait examiner par la suite.

Utilisation d'un saut non local

Une troisième manière de terminer un gestionnaire de signaux est d'utiliser un saut non local `siglongjmp()`. Dans ce cas, l'exécution reprend dans un contexte différent, qui a été sauvegardé auparavant. On évite ainsi certains risques de bogues dus à l'arrivée intempestive de signaux, tels que nous en utiliserons pour `SIGALRM` à la fin de ce chapitre. De même, cette méthode permet de reprendre le contrôle d'un programme qui a, par exemple, reçu un signal indiquant une instruction illégale. SUSv3 précise que le comportement d'un programme qui ignore les signaux d'erreur du type `SIGFPE`, `SIGILL`, `SIGSEGV` est indéfini. Nous avons vu que certaines de ces erreurs peuvent se produire à la suite de débordements de pile ou de mauvaises saisies de l'utilisateur dans le cas des routines mathématiques. Certaines applications désirent rester insensibles à ces erreurs et reprendre leur exécution comme si de rien n'était. C'est possible grâce à l'emploi de **`sigsetjmp()`** et **`siglongjmp()`**. Ces deux appels système sont des extensions des anciens **`setjmp()`** et **`longjmp()`**, qui posaient des problèmes avec les gestionnaires de signaux.

L'appel système `sigsetjmp()` a le prototype suivant, déclaré dans `<setjmp.h>` :

```
int sigsetjmp (sigjmp_buf contexte, int sauver_signaux);
```

Lorsque `sigsetjmp()` est invoqué dans le programme, il mémorise dans le buffer transmis en premier argument le contexte d'exécution et renvoie 0. Si son second argument est non nul, il mémorise également le masque de blocage des signaux dans le premier argument.

Lorsque le programme rencontre l'appel système `siglongjmp()`, dont le prototype est :

```
void siglongjmp (sigjmp_buf contexte, int valeur);
```

l'exécution reprend exactement à l'emplacement du `sigsetjmp()` correspondant au même buffer, et celui-ci renvoie alors la valeur indiquée en second argument de `siglongjmp()`. Cette valeur permet de différencier la provenance du saut, par exemple depuis plusieurs gestionnaires de signaux d'erreur.

L'inconvénient des sauts non locaux est qu'un usage trop fréquent diminue sensiblement la lisibilité des programmes. Il est conseillé de les réserver toujours au même type de circonstances dans une application donnée, pour gérer par exemple des temporisations, comme nous le verrons ultérieurement avec le signal `SIGALRM`.

Nous allons pour l'instant créer un programme qui permet à l'utilisateur de saisir deux valeurs numériques entières, et qui les divise l'une par l'autre. Si un signal `SIGFPE` se produit (on a demandé une division par zéro), l'exécution reprendra quand même dans un contexte propre.

exemple_siglongjmp.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <setjmp.h>

    sigjmp_buf      contexte;

    void
gestionnaire_sigfpe (int numero)
{
    siglongjmp(contexte, 1);
    /* Si on est ici le saut a raté, il faut quitter */
    signal(numero, SIG_DFL);
    raise(numero);
}

    int
main (void)
{
    int     p, q, r;

    struct sigaction action;
    action.sa_handler = gestionnaire_sigfpe;
    action.sa_flags = 0;
    sigfillset(& action.sa_mask);
    sigaction(SIGFPE, & action, NULL);

    while (1) {
        if (sigsetjmp(contexte, 1) != 0) {
            /* On est arrivé ici par siglongjmp() */
            fprintf(stdout, "Aie ! erreur mathématique ! \n");
            fflush(stdout);
        }
        while (1) {
            fprintf(stdout, "Entrez le dividende p : ");
            if (fscanf(stdin, "%d", & p) == 1)
                break;
        }
        while (1) {
            fprintf(stdout, "Entrez le diviseur q : ");
            if (fscanf(stdin, "%d", & q) == 1)
                break;
        }
        r = p / q;
        fprintf(stdout, "rapport p / q = %d\n", r);
    }

    return EXIT_SUCCESS;
}
```

Un petit exemple d'exécution :

```
$ ./exemple_siglongjmp
Entrez le dividende p : 8
Entrez le diviseur q : 2
rapport p / q = 4
Entrez le dividende p : 6
Entrez le diviseur q : 0
Aie ! erreur mathématique !
Entrez le dividende p : 6
Entrez le diviseur q : 3
rapport p / q = 2
Entrez le dividende p : (Contrôle-C)
$
```

Ce genre de technique est surtout utilisée dans les interpréteurs de langages comme Lisp pour permettre de revenir à une boucle principale en cas d'erreur.

Les anciens appels système `setjmp()` et `longjmp()` fonctionnaient de la même manière, mais ne sauvegardaient pas le masque des signaux bloqués (comme si le second argument de `siglongjmp()` valait 0). Le masque retrouvé dans le corps du programme n'est donc pas nécessairement celui qui est attendu. En effet, au sein d'un gestionnaire, le noyau bloque le signal concerné, ce qui n'est sûrement pas ce qu'on désire dans la boucle principale du programme.

Un signal particulier : l'alarme

Le signal `SIGALRM` est souvent utilisé comme temporisation pour indiquer un délai maximal d'attente pour des appels système susceptibles de bloquer. On utilise l'appel système **`alarm()`** pour programmer une temporisation avant la routine concernée, et `SIGALRM` sera déclenché lorsque le délai sera écoulé, faisant échouer l'appel bloquant avec le code `EINTR` dans `errno`. Si la routine se termine normalement avant le délai maximal, on annule la temporisation avec `alarm(0)`.

Il y a de nombreuses manières de programmer des temporisations, mais peu sont tout à fait fiables. On considérera que l'appel système à surveiller est une lecture depuis une socket réseau.

Il est évident que `SIGALRM` doit être intercepté par un gestionnaire installé avec `sigaction()` sans l'option `RESTART` dans `sa_flags` (sinon l'appel bloqué redémarrerait automatiquement). Ce gestionnaire peut être vide, son seul rôle est d'interrompre l'appel système lent.

```
    void
gestionnaire_sigalrm (int inutilise) {
    /* ne fait rien */
}
```

L'installation en est faite ainsi :

```
struct sigaction    action;

sigemptyset(& (action . sa_mask));
action.sa_flags = 0;
action.sa_handler = gestionnaire_sigalrm;
sigaction(SIGALRM, action, NULL);
```

Nous allons commencer par cet exemple naïf :

```
alarm(delai_maximal);
taille_lue = read(fd_socket, buffer, taille_buffer);
alarm(0);
if ((taille_lue != taille_buffer) && (errno == EINTR))
    fprintf(stderr, "délai maximal écoulé \n");
    return(-1);
}
/* ... suite ... */
```

SUSv3 autorisant l'appel système `read()` à renvoyer soit –1, soit le nombre d'octets lus lors d'une interruption par un signal, nous comparerons sa valeur de retour avec la taille attendue et non avec –1. Cela améliore la portabilité de notre programme.

Le premier problème qui se pose est qu'un signal autre que l'alarme peut avoir interrompu l'appel système `read()`. Cela peut se résoudre en imposant que tous les autres signaux gérés par le programme aient l'attribut `SA_RESTART` validé pour faire redémarrer l'appel bloquant. Toutefois, un problème subsiste, car le redémarrage n'a généralement lieu que si `read()` n'a pu lire aucun octet avant l'arrivée du signal. Sinon, l'appel se termine quand même en renvoyant le nombre d'octets lus.

Le second problème est que, sur un système très chargé, le délai peut s'écouler entièrement entre la programmation de la temporisation et l'appel système lui-même. Il pourrait alors rester bloqué indéfiniment.

Ce qu'on aimerait, c'est disposer d'un équivalent à `sigsuspend()`, qui permette d'effectuer de manière atomique le déblocage d'un signal et d'un appel système. Malheureusement, cela n'existe pas.

Nous allons donc utiliser une autre méthode, plus complexe, utilisant les sauts non locaux depuis le gestionnaire. Quel que soit le moment où le signal se déclenche, nous reviendrons au même emplacement du programme et nous annulerons alors la lecture. Bien entendu, le gestionnaire de signaux doit être modifié. Il n'a plus à être installé sans l'option `SA_RESTART` puisqu'il ne se terminera pas normalement.

Cet exemple va servir à temporiser la saisie d'une valeur numérique depuis le clavier. Nous lirons une ligne complète, puis nous essayerons d'y trouver un nombre entier. En cas d'échec, nous recommencerons. Malgré tout, un délai maximal de 5 secondes est programmé, après lequel le programme abandonne.

exemple_alarm.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <setjmp.h>
    #include <unistd.h>

    sigjmp_buf      contexte_sigalrm;

    void
gestionnaire_sigalrm (int inutilise)
{
    siglongjmp(contexte_sigalrm, 1);
```

```
}

    int
main (void)
{
    char    ligne[80];
    int     i;

    struct sigaction action;
    action.sa_handler = gestionnaire_sigalrm;
    action.sa_flags = 0;
    sigfillset(& action.sa_mask);

    sigaction(SIGALRM, & action, NULL);
    fprintf(stdout, "Entrez un nombre entier avant 5 secondes : ");
    if (sigsetjmp(contexte_sigalrm, 1) == 0) {
        /* premier passage, installation */
        alarm(5);

        /* Lecture et analyse de la ligne saisie */
        while (1) {
            if (fgets(ligne, 79, stdin) != NULL)
                if (sscanf(ligne, "%d", & i) == 1)
                    break;
            fprintf(stdout, "Un entier svp : ");
        }
        /* Ok - La ligne est bonne */
        alarm(0);
        fprintf(stdout, "Ok !\n");
    } else {
        /* On est arrivé par SIGALRM */
        fprintf(stdout, "\n Trop tard !\n");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Voici quelques exemples d'exécution :

```
$ ./exemple_alarm
Entrez un nombre entier avant 5 secondes : 6
Ok !
$ ./exemple_alarm
Entrez un nombre entier avant 5 secondes : a
Un entier svp : z
Un entier svp : e
Un entier svp : 8
Ok !
$ ./exemple_alarm
Entrez un nombre entier avant 5 secondes :
Trop tard !
$
```

Nous avons ici un exemple de gestion de délai fiable, fonctionnant avec n'importe quelle fonction de bibliothèque ou d'appel système risquant de rester bloqué indéfiniment. Le seul inconvénient de ce programme est le risque que le signal `SIGALRM` se déclenche alors que le processus est en train d'exécuter le gestionnaire d'un autre signal (par exemple `SIGUSR1`). Dans ce cas, on ne revient pas au gestionnaire interrompu et ce signal est perdu.

La seule possibilité pour l'éviter est d'ajouter systématiquement `SIGALRM` dans l'ensemble des signaux bloqués lors de l'exécution des autres gestionnaires, c'est-à-dire en l'insérant dans chaque champ `sa_mask` des signaux interceptés :

```
struct sigaction action;
action.sa_handler = gestionnaire_sigusr1;
action.sa_flags = SA_RESTART;
sigemptyset(& (action.sa_mask));
sigaddset(& (action.sa_mask), SIGALARM);
sigaction(SIGUSR1, & action, NULL);
```

Le signal `SIGALRM` n'interrompra alors jamais l'exécution complète du gestionnaire `SIGUSR1`.

Conclusion

Nous avons étudié dans les deux derniers chapitres l'essentiel de la programmation habituelle concernant les signaux. Certaines confusions interviennent parfois à cause d'appels système obsolètes, qu'on risque néanmoins de rencontrer encore dans certaines applications.

Des précisions concernant le comportement des signaux sur d'autres systèmes sont disponibles dans [STEVENS 1993] *Advanced Programming in the Unix Environment*. La programmation portable en utilisant les fonctions Posix (SUSv3 à présent) est décrite dans [LEWINE 1994] *Posix Programmer's Guide*.

Le prochain chapitre sera consacré à un aspect plus moderne des signaux, introduits dans le noyau Linux depuis sa version 2.2 : les signaux temps-réel.

8

Signaux temps-réel

Avec Linux 2.2 est apparue la gestion des signaux temps-réel. Ceux-ci constituent une extension des signaux `SIGUSR1` et `SIGUSR2`, qui présentaient trop de limitations pour des applications temps-réel. Il faut entendre, par le terme temps-réel, une classe de programmes pour lesquels le temps mis pour effectuer une tâche constitue un facteur important du résultat. Une application temps-réel n'a pas forcément besoin d'être très rapide ni de répondre dans des délais très brefs, mais simplement de respecter des limites temporelles connues.

Ceci est bien entendu contraire à tout fonctionnement multitâche préemptif, puisque aucune garantie de temps de réponse n'est fournie par le noyau. Nous verrons alors qu'il est possible de commuter l'ordonnancement des processus pour obtenir un séquencement beaucoup plus proche d'un véritable support temps-réel. Nous reviendrons sur ces notions dans le chapitre 11.

Les fonctionnalités temps-réel pour les systèmes Unix furent introduites par la norme Posix.1b et sont décrites de nos jours par l'option *RTS* de SUSv3 ; leur support par Linux élargit le champ des applications industrielles et scientifiques utilisables sur ce système d'exploitation.

Les signaux temps-réel présentent donc les caractéristiques suivantes par rapport aux signaux classiques :

- nombre plus important de signaux utilisateur ;
- empilement des occurrences des signaux bloqués ;
- délivrance prioritaire des signaux ;
- informations supplémentaires fournies au gestionnaire.

Caractéristiques des signaux temps-réel

Nombre de signaux temps-réel

Nous avions déjà remarqué que le fait de ne disposer que de deux signaux réservés au programmeur était une contrainte importante pour le développement d'applications utilisant beaucoup cette méthode de communication.

La norme Posix.1b réclame la présence d'au moins huit signaux temps-réel. Linux en propose trente-deux, ce qui est largement suffisant pour la plupart des situations.

> **Attention**
> Certaines implémentations d'Unix (Solaris par exemple) sont véritablement limitées à huit signaux temps-réel, il faudra donc être économe pour les applications voulues portables.

Les signaux temps-réel n'ont pas de noms spécifiques, contrairement aux signaux classiques. On peut employer directement leurs numéros, qui s'étendent de `SIGRTMIN` à `SIGRTMAX` compris. Bien entendu, on utilisera des positions relatives dans cet intervalle, par exemple (`SIGRTMIN + 5`) ou (`SIGRTMAX - 2`), sans jamais préjuger de la valeur effective de ces constantes.

Il est de surcroît conseillé, pour améliorer la qualité du code source, de définir des constantes symboliques pour nommer les signaux utilisés dans le code. Par exemple, on définira dans un fichier d'en-tête de l'application des constantes :

```
#define SIGRT0    (SIGRTMIN)
#define SIGRT1    (SIGRTMIN + 1)
#define SIGRT2    (SIGRTMIN + 2)
```

ou, encore mieux, des constantes dont les noms soient parlants :

```
#define SIG_AUTOMATE_PRET      (SIGRTMIN + 2)
#define SIG_ANTENNE_AU_NORD    (SIGRTMIN + 4)
#define SIG_LIAISON_ETABLIE    (SIGRTMIN + 1)
```

On vérifiera également que le nombre de signaux temps-réel soit suffisant pour l'application. Toutefois, les valeurs `SIGRTMIN` et `SIGRTMAX` peuvent être implémentées sous forme de variables, et pas de constantes symboliques. Cette vérification doit donc avoir lieu durant l'exécution du programme, pas pendant sa compilation. On emploiera ainsi un code du genre :

```
#include <signal.h>
#include <unistd.h>

#ifndef _POSIX_REALTIME_SIGNALS
    #error "Pas de signaux temps-réel disponibles"
#endif

#define SIGRT0    (SIGRTMIN)
[...]
#define SIGRT10   (SIGRTMIN + 10)

#define NB_SIGRT_UTILES    11
```

```
    int
main (int argc, char ** argv [])
{
    if ((SIGRTMAX - SIGRTMIN + 1)< NB_SIGRT_UTILES) {
        fprintf(stderr, "Pas assez de signaux temps-réel \n");
        exit(EXIT_FAILURE);
    }
    [...]
}
```

Empilement des signaux bloqués

Nous avons vu que les signaux classiques ne sont pas empilés. Cela signifie que si deux occurrences d'un même signal arrivent alors que celui-ci est temporairement bloqué, une seule d'entre elles sera finalement délivrée au processus lors du déblocage. Rappelons que le blocage n'intervient pas nécessairement de manière explicite, mais peut aussi se produire simplement durant l'exécution du gestionnaire d'un autre signal.

Lorsqu'on veut s'assurer qu'un signal arrivera effectivement à un processus, il faut mettre au point un système d'acquittement, compliquant sérieusement le code.

Comme un signal est automatiquement bloqué durant l'exécution de son propre gestionnaire, une succession à court intervalle de trois occurrences consécutives du même signal risque de faire disparaître la troisième impulsion. Ce comportement n'est pas acceptable dès qu'un processus doit assurer des comptages ou des commutations d'état.

Pour pallier ce problème, la norme Posix.1b a introduit la notion d'empilement des signaux bloqués. Si un signal bloqué est reçu quatre fois au niveau d'un processus, nous sommes sûr qu'il sera délivré quatre fois lors de son déblocage.

Il existe bien entendu une limite au nombre de signaux pouvant être mémorisés simultanément. Cette limite n'est pas précisée par SUSv3. Sous Linux, on peut empiler 1 024 signaux par processus, à moins que la mémoire disponible ne soit pas suffisante. L'appel-système `sigqueue()`, que nous verrons plus bas et qui remplace `kill()` pour les signaux temps-réel, permet d'avoir la garantie que le signal est bien empilé.

Délivrance prioritaire des signaux

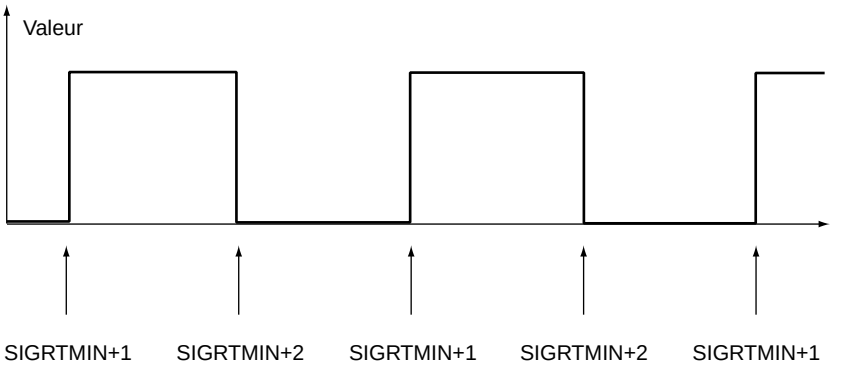
Lorsque le noyau a le choix entre plusieurs signaux temps-réel à transmettre au processus (par exemple lors d'un déblocage d'un ensemble complet), il délivre toujours les signaux de plus faible numéro en premier.

Les occurrences de `SIGRTMIN` seront donc toujours transmises en premier au processus, et celles de `SIGRTMAX` en dernier. Cela permet de gérer des priorités entre les événements représentés par les signaux. Par contre, SUSv3 ne donne aucune indication sur les priorités des signaux classiques. En général, ils sont délivrés avant les signaux temps-réel car ils indiquent pour le plupart des dysfonctionnements à traiter en urgence (`SIGSEGV`, `SIGILL`, `SIGHUP`…), mais nous n'avons aucune garantie concernant ce comportement.

La notion de priorité entre signaux peut néanmoins présenter un inconvénient si on n'y prend pas garde. Le revers de la médaille, c'est que les signaux ne sont plus indépendants, comme l'étaient `SIGUSR1` et `SIGUSR2`, par exemple. On pourrait vouloir utiliser deux signaux temps-réel pour implémenter un mécanisme de bascule, un signal (disons `SIGRTMIN+1`) demandant

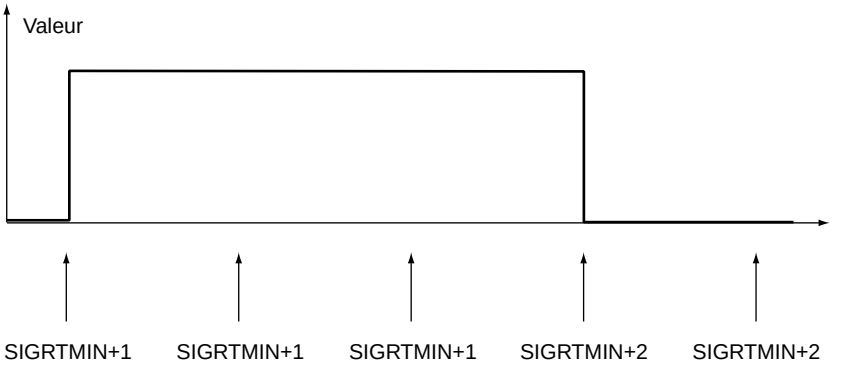
le passage à l'état 1, et l'autre (`SIGRTMIN+2`) la descente au niveau 0. On aurait alors une séquence représentée sur la figure suivante :

Figure 8.1
Séquence attendue



Malheureusement, si les signaux sont bloqués pendant un moment, ils ne seront pas délivrés dans l'ordre d'arrivée, mais en fonction de leur priorité. Toutes les impulsions `SIGRTMIN+1` sont délivrées d'abord, puis toutes les impulsions `SIGRTMIN+2`.

Figure 8.2
Séquence obtenue



Si des événements liés doivent être transmis à l'aide des signaux temps-réel, il faut se tourner vers une autre méthode, en utilisant un seul signal, mais en transmettant une information avec le signal lui-même.

Informations supplémentaires fournies au gestionnaire

Les signaux temps-réel sont capables de transmettre une – petite – quantité d'information au gestionnaire associé. Cette information est contenue dans une valeur de type `union sigval`. Cette union peut prendre deux formes :

- un entier `int`, en employant son membre `sigval_int` ;
- un pointeur `void *`, avec le membre `sigval_ptr`.

Nous avons déjà évoqué la forme du gestionnaire de signaux temps-réel dans le chapitre précédent, dans le paragraphe traitant de l'attribut `SA_SIGINFO` dans le champ `sa_flags` de `sigaction`.

```
void gestionnaire_signal (int numero,
                          struct siginfo * info,
                          void * inutile);
```

Le troisième argument de cette routine n'est pas défini de manière portable. Certains systèmes Unix l'utilisent, mais apparemment le noyau Linux n'en fait pas usage. Toutes les informations supplémentaires se trouvent dans la structure `siginfo` sur laquelle un pointeur est transmis en deuxième argument.

Pour que ce gestionnaire soit installé, il faut le placer dans le membre `sa_sigaction` de la structure `sigaction`, et non plus dans le membre `sa_handler`. De même, le champ `sa_flags` doit contenir l'attribut `SA_SIGINFO`.

L'initialisation se fait donc ainsi :

```
struct sigaction action;

action.sa_sigaction = gestionnaire_signal_temps_reel;
sigemptyset(& action . sa_mask);
action.sa_flags = SA_SIGINFO;
if (sigaction(SIGRTMIN + 1, & action, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
```

Émission d'un signal temps-réel

Bien sûr, si on désire transmettre des données supplémentaires au gestionnaire de signaux, il ne suffit plus d'employer la fonction `kill()` habituelle. Il existe un nouvel appel-système, nommé **`sigqueue()`**, défini par SUSv3 :

```
int sigqueue (pid_t pid, int numero, const union sigval valeur)
```

Les deux premiers arguments sont équivalents à ceux de `kill()`, mais le troisième correspond au membre `si_sigval` de la structure `siginfo` transmise au gestionnaire de signaux.

Il n'y a aucun moyen dans le gestionnaire de déterminer si l'argument de type `union sigval` a été rempli, lors de l'invocation de `sigqueue()` avec une valeur entière (champ `sigval_int`) ou un pointeur (champ `sigval_ptr`). Il est donc nécessaire que l'application reste cohérente entre l'envoi du signal et sa réception. Lorsque le signal est transmis entre deux processus distincts, on ne peut bien sûr passer que des valeurs entières, un pointeur n'ayant pas de signification dans l'espace d'adressage de l'autre processus.

> Il y a peu de situations où le passage d'un pointeur associé à un signal se justifie. On peut penser aux threads qui partagent leur espace mémoire, mais également aux cas où le processus s'envoie lui-même un signal différé. Nous verrons dans le chapitre 30 que l'on peut programmer une opération d'entrée-sortie asynchrone (par exemple une écriture en arrière-plan) et être averti lorsqu'elle est terminée par l'arrivée d'un signal afin de vérifier que tout s'est bien passé. Dans cette situation on peut associer au signal un pointeur sur une structure de données décrivant l'opération afin d'avoir toutes les informations nécessaires dans le gestionnaire invoqué.

Récapitulons les principaux champs de la structure `siginfo` reçue par le gestionnaire de signaux :

Nom membre	Type	Posix.1b	Signification
`si_signo`	`int`	•	Numéro du signal, redondant avec le premier argument de l'appel du gestionnaire.
`si_code`	`int`	•	Voir ci-dessous.
`si_value.sigval_int`	`int`	•	Entier de l'union passée en dernier argument de `sigqueue()`.
`si_value.sigval_ptr`	`void *`	•	Pointeur de l'union passée en dernier argument de `sigqueue()`. Ne doit pas être employé simultanément avec le membre précédent.
`si_errno`	`int`		Valeur de la variable globale `errno` lors du déclenchement du gestionnaire. Permet de rétablir cette valeur en sortie.
`si_pid`	`pid_t`		PID du processus fils s'étant terminé si le signal est SIGCHLD. PID de l'émetteur si le signal est temps-réel.
`si_uid`	`uid_t`		UID réel de l'émetteur d'un signal temps-réel ou celui du processus fils terminé si le signal est SIGCHLD.
`si_status`	`int`		Code de retour du processus fils terminé, uniquement avec le signal SIGCHLD.

La signification du champ `si_code` varie suivant le type de signal. Pour les signaux temps-réel ou pour la plupart des signaux classiques, `si_code` indique l'origine du signal :

Valeur	Provenance du signal	Posix.1b
`SI_KERNEL`	Signal émis par le noyau	
`SI_USER`	Appel-système `kill()` ou `raise()`	•
`SI_QUEUE`	Appel-système `sigqueue()`	•
`SI_ASYNCIO`	Terminaison d'une entrée-sortie asynchrone	•
`SI_MESGQ`	Changement d'état d'une file de message temps-réel (non implémenté sous Linux)	•
`SI_SIGIO`	Changement d'état sur un descripteur d'entrée-sortie asynchrone	
`SI_TIMER`	Expiration d'une temporisation temps-réel (non implémentée sous Linux)	•

Pour un certain nombre de signaux classiques, Linux fournit également des données (principalement utiles au débogage) dans le champ `si_code`, si le gestionnaire est installé en utilisant `SA_SIGINFO` dans l'argument `sa_flags` de `sigaction` :

Signal SIGBUS	
`BUS_ADRALN`	Erreur d'alignement d'adresse.
`BUS_ADRERR`	Adresse physique invalide.
`BUS_OBJERR`	Erreur d'adressage matériel.

Signal SIGCHLD	
CLD_CONTINUED	Un fils arrêté a redémarré (ce code n'est effectivement implémenté dans le noyau que depuis Linux 2.6.9).
CLD_DUMPED	Un fils s'est terminé anormalement.
CLD_EXITED	Un fils vient de se terminer normalement.
CLD_KILLED	Un fils a été tué par un signal.
CLD_STOPPED	Un fils a été arrêté.
CLD_TRAPPED	Un fils a atteint un point d'arrêt.
Signal SIGFPE	
FPE_FLTDIV	Division en virgule flottante par zéro.
FPE_FLTINV	Opération en virgule flottante invalide.
FPE_FLTOVF	Débordement supérieur lors d'une opération en virgule flottante.
FPE_FLTRES	Résultat faux lors d'une opération en virgule flottante.
FPE_FLTSUB	élévation à une puissance invalide.
FPE_FLTUND	Débordement inférieur lors d'une opération en virgule flottante.
FPE_INTDIV	Division entière par zéro.
FPE_INTOVF	Débordement de valeur entière.
Signal SIGILL	
ILL_BADSTK	Erreur de pile.
ILL_COPROC	Erreur d'un coprocesseur.
ILL_ILLADR	Mode d'adressage illégal.
ILL_ILLOPC	Code d'opération illégal.
ILL_ILLOPN	Opérande illégale.
ILL_ILLTRP	Point d'arrêt illégal.
ILL_PRVOPC	Code d'opération privilégié.
ILL_PRVREG	Accès à un registre privilégié.
Signal SIGPOLL	
POLL_ERR	Erreur d'entrée-sortie.
POLL_HUP	Déconnexion du correspondant.
POLL_IN	Données prêtes à être lues.
POLL_MSG	Message disponible en entrée.
POLL_OUT	Zone de sortie disponible.
POLL_PRI	Entrées disponibles à haute priorité.
Signal SIGSEGV	
SEGV_ACCERR	Accès interdit à la projection mémoire.
SEGV_MAPERR	Adresse sans projection mémoire.
Signal SIGTRAP	
TRAP_BRKPT	Point d'arrêt de débogage.
TRAP_TRACE	Point d'arrêt de profilage.

Attention

Tous ces codes sont spécifiques à Linux et ne doivent pas être employés dans une application portable. En outre, ils sont tous déclarés dans les fichiers d'en-tête de Linux, mais ils ne sont pas tous réellement renvoyés par le noyau.

À la lecture du premier tableau, concernant les champs `si_code` généraux, nous remarquons plusieurs choses :

- Il est possible d'envoyer un signal temps-réel avec l'appel-système `kill()`. Simplement, les informations supplémentaires ne seront pas disponibles. Leur valeur dans ce cas n'est pas précisée par SUSv3, mais sous Linux, le champ de type `sigval` correspondant est mis à zéro. Il est donc possible d'employer les signaux temps-réel en remplacement pur et simple de SIGUSR1 et SIGUSR2 dans une application déjà existante, en profitant de l'empilement des signaux, mais en restant conscient du problème que nous avons évoqué, concernant la priorité de délivrance.
- Il existe un certain nombre de sources de signaux temps-réel possibles, en supplément de la programmation manuelle avec `sigqueue()` ou `kill()`. Plusieurs fonctionnalités introduites par la norme Posix.1b permettent en effet à l'application de programmer un travail et de recevoir un signal lorsqu'il est accompli. C'est le cas, par exemple, des files de messages utilisant les fonctions `mq_open()`, `mq_close()`, `mq_notify()`, ou encore des temporisations programmées avec `timer_create()`, `timer_delete()` et `timer_settime()`.

Nous allons commencer par créer un programme servant de frontal à `sigqueue()`, comme l'utilitaire système `/bin/kill` pouvait nous servir à invoquer l'appel-système `kill()` depuis la ligne de commande.

exemple_sigqueue.c :

```
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    void
syntaxe (const char * nom)
{
    fprintf(stderr, "syntaxe %s signal pid...\n", nom);
    exit(EXIT_FAILURE);
}

    int
main (int argc, char * argv [])
{
    int     i;
    int     numero;
    int     pid;
    union sigval valeur;

    if (argc == 1)
        syntaxe(argv[0]);
```

```
    i = 1;
    if (argc == 2) {
        numero = SIGTERM;
    } else {
        if (sscanf(argv[i], "%d", & numero) != 1)
            syntaxe(argv[0]);
        i ++;
    }
    if ((numero < 0) || (numero > NSIG - 1))
        syntaxe(argv[0]);
    valeur.sival_int = 0;
    for (; i < argc; i ++) {
        if (sscanf(argv[i], "%d", & pid) != 1)
            syntaxe(argv[0]);
        if (sigqueue((pid_t) pid, numero, valeur) < 0) {
            fprintf(stderr, "%d : ", pid);
            perror ("");
        }
    }
    return EXIT_SUCCESS;
}
```

À présent, nous allons créer un programme qui installe un gestionnaire de type temps-réel pour tous les signaux – même les signaux classiques – pour afficher le champ `si_code` de leur argument de type `siginfo`.

exemple_siginfo.c :

```
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    void
gestionnaire (int numero, struct siginfo * info, void * inutilise)
{
    fprintf(stderr, "Reçu %d\n", numero);
    fprintf(stderr, "  si_code = %d\n", info->si_code);
}

    int
main (void)
{
    int              i;
    struct sigaction action;
    char             chaine[5];

    action.sa_sigaction = gestionnaire;
    action.sa_flags = SA_SIGINFO;
    sigemptyset(& action.sa_mask);
    fprintf(stderr, "PID=%ld\n", (long) getpid());
    for (i = 1; i < NSIG; i ++)
```

```
        if (sigaction(i, & action, NULL) < 0)
            fprintf(stderr, "%d non intercepté \n", i);
    while (1)
        fgets(chaine, 5, stdin);
    return EXIT_SUCCESS;
}
```

Finalement, nous lançons le programme `exemple_siginfo`, puis nous lui envoyons des signaux depuis une autre console (représentée en seconde colonne), en utilisant tantôt `/bin/kill`, tantôt `exemple_sigqueue`.

```
$ ./exemple_siginfo
PID=1069
9 non intercepté
19 non intercepté
                            $ kill -33 1069
Reçu 33
  si_code = 0
                            $ ./exemple_sigqueue 33 1069
Reçu 33
  si_code = -1
                            $ kill -TERM 1069
Reçu 15
  si_code = 0
                            $ kill -KILL 1069
Killed
$
```

Le champ `si_code` correspond bien à 0 (valeur de `SI_USER`) ou à –1 (valeur de `SI_QUEUE`) suivant le cas.

Attention

Si on utilise l'appel-système `alarm()`pour déclencher `SIGALRM`, le champ `si_code` est rempli avec la valeur `SI_USER` et pas avec `SI_TIMER`, qui est réservée aux temporisations temps-réel.

Notre second exemple va mettre en évidence à la fois l'empilement des signaux temps-réel et leur respect d'une priorité. Notre programme va en effet bloquer tous les signaux, s'en envoyer une certaine quantité, et voir dans quel ordre ils arrivent. La valeur `sigval` associée aux signaux permettra de les reconnaître.

exemple_sigqueue_1.c :

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int   signaux_arrives[10];
int   valeur_arrivee[10];
int   nb_signaux = 0;
```

```
    void
gestionnaire_signal_temps_reel (int numero,
                siginfo_t * info, void * inutile)
{
    signaux_arrives[nb_signaux] = numero - SIGRTMIN;
    valeur_arrivee[nb_signaux]  = info->si_value.sival_int;
    nb_signaux ++;
}

    void
envoie_signal_temps_reel (int numero, int valeur)
{
    union sigval valeur_sig;

    fprintf(stdout, "Envoi signal SIRTMIN+%d, valeur %d\n",
                            numero, valeur);
    valeur_sig.sival_int = valeur;
    if (sigqueue(getpid(), numero + SIGRTMIN, valeur_sig) < 0) {
        perror("sigqueue");
        exit(EXIT_FAILURE);
    }
}

    int
main (void)
{
    struct sigaction action;
    sigset_t         ensemble;
    int              i;

    fprintf(stdout, "Installation gestionnaires de signaux \n");
    action.sa_sigaction = gestionnaire_signal_temps_reel;
    sigemptyset(& action.sa_mask);
    action.sa_flags = SA_SIGINFO;
    if ((sigaction(SIGRTMIN + 1, & action, NULL) < 0)
     || (sigaction(SIGRTMIN + 2, & action, NULL) < 0)
     || (sigaction(SIGRTMIN + 3, & action, NULL) < 0)) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Blocage de tous les signaux \n");
    sigfillset(& ensemble);
    sigprocmask(SIG_BLOCK, & ensemble, NULL);

    envoie_signal_temps_reel(1, 0);
    envoie_signal_temps_reel(2, 1);
    envoie_signal_temps_reel(3, 2);
    envoie_signal_temps_reel(1, 3);
    envoie_signal_temps_reel(2, 4);
    envoie_signal_temps_reel(3, 5);
    envoie_signal_temps_reel(3, 6);
    envoie_signal_temps_reel(2, 7);
```

```
    envoie_signal_temps_reel(1, 8);
    envoie_signal_temps_reel(3, 9);

    fprintf(stdout, "Déblocage de tous les signaux \n");
    sigfillset(& ensemble);
    sigprocmask(SIG_UNBLOCK, & ensemble, NULL);

    fprintf(stdout, "Affichage des résultats \n");
    for (i = 0; i < nb_signaux; i ++)
        fprintf(stdout, "Signal SIGRTMIN+%d, valeur %d\n",
                        signaux_arrives[i], valeur_arrivee[i]);

    fprintf(stdout, "Fin du programme \n");
    return EXIT_SUCCESS;
}
```

Notre gestionnaire stocke les signaux arrivant dans une table qui est affichée par la suite, pour éviter les problèmes de concurrence sur l'accès au flux `stdout`.

```
$ ./exemple_sigqueue_1
Installation gestionnaires de signaux
Blocage de tous les signaux
Envoi signal SIRTMIN+1, valeur 0
Envoi signal SIRTMIN+2, valeur 1
Envoi signal SIRTMIN+3, valeur 2
Envoi signal SIRTMIN+1, valeur 3
Envoi signal SIRTMIN+2, valeur 4
Envoi signal SIRTMIN+3, valeur 5
Envoi signal SIRTMIN+3, valeur 6
Envoi signal SIRTMIN+2, valeur 7
Envoi signal SIRTMIN+1, valeur 8
Envoi signal SIRTMIN+3, valeur 9
Déblocage de tous les signaux
Affichage des résultats
Signal SIGRTMIN+1, valeur 0
Signal SIGRTMIN+1, valeur 3
Signal SIGRTMIN+1, valeur 8
Signal SIGRTMIN+2, valeur 1
Signal SIGRTMIN+2, valeur 4
Signal SIGRTMIN+2, valeur 7
Signal SIGRTMIN+3, valeur 2
Signal SIGRTMIN+3, valeur 5
Signal SIGRTMIN+3, valeur 6
Signal SIGRTMIN+3, valeur 9
Fin du programme
$
```

Nous remarquons bien que les signaux sont délivrés suivant leur priorité : tous les `SIRTMIN+1` en premier, suivis des `SIGRTMIN+2`, puis des `SIGRTMIN+3`. De même, au sein de chaque classe, les occurrences des signaux sont bien empilées et délivrées dans l'ordre chronologique d'émission.

Traitement rapide des signaux temps-réel

La norme SUSv3 donne accès à des possibilités de traitement rapide des signaux. Ceci ne concerne que les applications qui attendent passivement l'arrivée d'un signal pour agir. Cette situation est assez courante lorsqu'on utilise les signaux comme une méthode pour implémenter un comportement multitâche au niveau applicatif.

Avec le traitement classique des signaux, nous utilisions quelque chose comme :

```
sigfillset(& tous_signaux);
sigprocmask(SIG_BLOCK, & tous_signaux, NULL);
sigemptyset(& aucun_signal);
while (! fin_programme)
    sigsuspend(& aucun_signal);
```

Lorsqu'un signal arrive, le processus doit alors être activé par l'ordonnanceur, ensuite l'exécution est suspendue, le gestionnaire de signaux est invoqué, puis le contrôle revient au fil courant d'exécution, qui termine la fonction `sigsuspend()`. La boucle reprend alors.

Le problème est que l'appel du gestionnaire par le noyau nécessite un changement de contexte, tout comme le retour de ce gestionnaire. Ceci est plus coûteux qu'un simple appel de fonction usuel.

Deux appels-système, définis dans la norme Posix.1b, ont donc fait leur apparition avec le noyau Linux 2.2. Il s'agit de **`sigwaitinfo()`** et de **`sigtimedwait()`**. Ce sont en quelque sorte des extensions de `sigsuspend()`. Ils permettent d'attendre l'arrivée d'un signal dans un ensemble précis. À la différence de `sigsuspend()`, lorsqu'un signal arrive, son gestionnaire n'est pas invoqué. À la place, l'appel-système `sigwaitinfo()` ou `sigtimedwait()` se termine en renvoyant le numéro de signal reçu. Il n'est plus nécessaire d'effectuer des changements de contexte pour exécuter le gestionnaire, il suffit d'une gestion directement intégrée dans le fil du programme (la plupart du temps en utilisant une construction `switch-case`). Si le processus doit appeler le gestionnaire, il le fera simplement comme une fonction classique, avec toutes les possibilités habituelles d'optimisation par insertion du code en ligne.

Comme prévu, `sigtimedwait()` est une version temporisée de `sigwaitinfo()`, qui échoue avec une erreur `EAGAIN` si aucun signal n'est arrivé pendant le délai imparti :

```
int sigwaitinfo  (const sigset_t * signaux_attendus, siginfo_t * info);
int sigtimedwait (const sigset_t * signaux_attendus, siginfo_t * info,
                  const struct timespec * delai);
```

De plus, ces fonctions offrent l'accès aux données supplémentaires disponibles avec les signaux temps-réel.

Attention

`sigsuspend()` prenait en argument l'ensemble des signaux bloqués, `sigwaitinfo()` comme `sigtimedwait()` réclament l'ensemble des signaux attendus.

La structure `timespec` utilisée pour programmer le délai offre les membres suivants :

Type	Nom	Signification
`long`	`tv_sec`	Nombre de secondes
`long`	`tv_nsec`	Nombre de nanosecondes

La valeur du champ `tv_nsec` doit être comprise entre 0 et 999.999.999, sinon le comportement est indéfini.

Les appels-système `sigwaitinfo()` ou `sigtimedwait()` peuvent échouer avec l'erreur `EINTR` si un signal non attendu est arrivé et a été traité par un gestionnaire. Si leur second argument est `NULL`, aucune information n'est stockée.

Il est important de bloquer avec `sigprocmask()` les signaux qu'on attend avec `sigwaitinfo()` ou `sigtimedwait()`, car cela assure qu'aucun signal impromptu n'arrivera juste avant ou après l'invocation de l'appel-système.

Notre premier exemple va consister à installer un gestionnaire normal pour un seul signal, `SIGRTMIN+1`, pour voir le comportement du système avec les signaux non attendus. Ensuite, on bloque tous les signaux, puis on les attend tous, sauf `SIGRTMIN+1` et `SIGKILL`. Nous expliquerons plus bas pourquoi traiter `SIGKILL` spécifiquement.

exemple_sigwaitinfo.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <unistd.h>

    void
gestionnaire (int numero, struct siginfo * info, void * inutile)
{
    fprintf(stderr, "gestionnaire : %d reçu \n", numero);
}

    int
main (void)
{
    sigset_t         ensemble;
    int              numero;
    struct sigaction action;
    fprintf(stderr, "PID=%u\n", getpid());
    /* Installation gestionnaire pour SIGRTMIN+1 */
    action.sa_sigaction = gestionnaire;
    action.sa_flags = SA_SIGINFO;
    sigemptyset(& action. sa_mask);
    sigaction(SIGRTMIN + 1, & action, NULL);
    /* Blocage de tous les signaux sauf SIGRTMIN+1 */
    sigfillset(& ensemble);
    sigdelset(& ensemble, SIGRTMIN + 1);
    sigprocmask(SIG_BLOCK, & ensemble, NULL);
```

```
    /* Attente de tous les signaux sauf RTMIN+1 et SIGKILL */
    sigfillset(& ensemble);
    sigdelset(& ensemble, SIGRTMIN + 1);
    sigdelse (& ensemble, SIGKILL);
    while (1) {
        if ((numero = sigwaitinfo(& ensemble, NULL)) < 0)
            perror("sigwaitinfo");
        else
            fprintf(stderr, "sigwaitinfo : %d reçu \n", numero);
    }
    return EXIT_SUCCESS;
}
```

Nous ne traitons pas réellement les signaux reçus, nous contentant d'afficher leur numéro, mais nous pourrions très bien insérer une séquence `switch-case` au retour de `sigwaitinfo()`. Il faut bien comprendre que le signal dont le numéro est renvoyé par `sigwaitinfo()` est complètement éliminé de la liste des signaux en attente. La structure `siginfo` est également remplie lors de l'appel-système si des informations sont disponibles.

Voici un exemple d'exécution avec, en seconde colonne, les actions saisies depuis une autre console :

```
$ ./exemple_sigwaitinfo
PID=1435
                                        $ kill -HUP 1435
sigwaitinfo : 1 reçu
                                        $ kill -TERM 1435
sigwaitinfo : 15 reçu
                                        $ kill -33 1435
gestionnaire : 33 reçu
sigwaitinfo: Appel système interrompu
                                        $ kill -KILL 1435
Killed
$
```

Nous remarquons que lorsqu'un signal non attendu est reçu, il est traité normalement par son gestionnaire, et l'appel-système `sigwaitinfo()` est interrompu et échoue avec l'erreur `EINTR`.

Conclusion

Nous avons examiné dans ce chapitre une extension importante des méthodes de traitement des signaux. Ces fonctionnalités temps-réel ajoutent une dimension considérable aux capacités de Linux à traiter des problèmes industriels ou scientifiques avec des délais critiques.

La norme Posix.1b, quand elle s'appelait encore Posix.4, a été étudiée en détail dans [GALLMEISTER 1995] *Posix.4 Programming for the real world*.

9

Sommeil des processus et contrôle des ressources

Nous allons étudier dans la première partie de ce chapitre les méthodes permettant d'endormir un processus pendant une durée plus ou moins précise. Nous aborderons ensuite les moyens de suivre l'exécution d'un programme et d'obtenir des informations statistiques le concernant. Naturellement, nous examinerons aussi les fonctions de limitation des ressources, permettant de restreindre l'utilisation du système par un processus.

Endormir un processus

La fonction la plus simple pour endormir temporairement un processus est **`sleep()`**, qui est déclarée ainsi dans `<unistd.h>` :

```
unsigned int sleep (unsigned int nb_secondes);
```

Cette fonction endort le processus pendant la durée demandée et revient ensuite. À cause de la charge du système, il peut arriver que `sleep()` dure un peu plus longtemps que prévu. De même, si un signal interrompt le sommeil du processus, la fonction `sleep()` revient plus tôt que prévu, en renvoyant le nombre de secondes restantes sur la durée initiale.

Notez que `sleep()` est une fonction de bibliothèque, qui n'est donc pas concernée par l'attribut `SA_RESTART` des gestionnaires de signaux, qui ne sert à relancer que les appels-système lents.

Voici un exemple dans lequel deux processus exécutent un appel à `sleep()`. Le processus père dort deux secondes avant d'envoyer un signal à son fils. Ce dernier essaye de dormir 10 secondes, mais sera réveillé plus tôt par le signal. On invoque la commande système « `date` » pour afficher l'heure avant et après l'appel `sleep()`. On présente également la durée restante :

exemple_sleep.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <unistd.h>
    #include <sys/wait.h>

    void
gestionnaire_sigusr1(int numero)
{
}

    int
main (void)
{
    pid_t            pid;
    unsigned int     duree_sommeil;
    struct sigaction action;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erreur dans fork \n");
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire_sigusr1;
    sigemptyset(& action.sa_mask);
    action.sa_flags = SA_RESTART;

    if (sigaction(SIGUSR1, & action, NULL) != 0) {
        fprintf(stderr, "Erreur dans sigaction \n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        system("date +\"%H:%M:%S\"");
        duree_sommeil = sleep(10);
        system("date +\"%H:%M:%S\"");
        fprintf(stdout, "Durée restante %u\n", duree_sommeil);
    } else {
        sleep(2);
        kill(pid, SIGUSR1);
        waitpid(pid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

Voici un exemple d’exécution :

```
$ ./exemple_sleep
12:31:19
12:31:21
Durée restante 8
$
```

La fonction `sleep()` étant implémentée à partir de l'appel-système `alarm()`, il est vraiment déconseillé de les utiliser ensemble dans la même portion de programme. La bibliothèque GlibC implémente `sleep()` en prenant garde aux éventuelles interactions avec une alarme déjà programmée, mais ce n'est pas forcément le cas sur d'autres systèmes sur lesquels on peut être amené à porter le programme.

De même, si un signal arrive pendant la période de sommeil et si le gestionnaire de ce signal modifie le comportement du processus vis-à-vis de `SIGALRM`, le résultat est totalement imprévisible. Également, si le gestionnaire de signaux se termine par un saut non local `siglongjmp()`, le sommeil est définitivement interrompu.

Lorsqu'on désire assurer une durée de sommeil assez précise malgré le risque d'interruption par un signal, on pourrait être tenté de programmer une boucle du type :

```
    void
sommeil (unsigned int duree_initiale)
{
    unsigned int duree_restante = duree_initiale;

    while (duree_restante > 0)
        duree_restante = sleep(duree_restante);
}
```

Malheureusement, ceci ne fonctionne pas, car lors d'une invocation de la fonction `sleep()` si un signal se produit au bout d'un dixième de seconde par exemple, la durée renvoyée sera quand même décrémentée d'une seconde complète. Si ce phénomène se produit à plusieurs reprises, un décalage certain peut se produire en fin de compte. Pour l'éviter, il faut recadrer la durée de sommeil régulièrement. On peut par exemple utiliser l'appel-système **`time()`**, qui est défini dans `<time.h>` ainsi :

```
time_t time (time_t * t);
```

Cet appel-système renvoie l'heure actuelle, sous forme du nombre de secondes écoulées depuis le 1er janvier 1970 à 0 heure GMT. De plus, si `t` n'est pas un pointeur `NULL`, cette valeur y est également stockée. Le format `time_t` est compatible avec un `unsigned long`. Nous reviendrons sur les fonctions de traitement du temps dans le chapitre 25.

Voici un exemple de routine de sommeil avec une durée précise :

```
    void
sommeil (unsigned int duree_initiale)
{
    time_t    heure_fin;
    time_t    heure_actuelle;

    heure_fin = time(NULL) + duree_initiale;

    while ((heure_actuelle = time(NULL)) < heure_fin)
        sleep(heure_fin - heure_actuelle);
}
```

Cette routine peut quand même durer un peu plus longtemps que prévu si le système est très chargé, mais elle restera précise sur des longues durées, même si de nombreux signaux sont reçus par le processus.

Si on désire avoir une résolution plus précise que la seconde, la fonction **usleep()** est disponible. Il faut imaginer que le « u » représente en réalité le « µ » de microseconde. Le prototype de cette fonction est déclaré dans <unistd.h> ainsi :

```
void usleep (unsigned long nb_micro_secondes);
```

L'appel endort le processus pendant le nombre indiqué de microsecondes, à moins qu'un signal ne soit reçu entre-temps. La fonction usleep() ne renvoyant pas de valeur, on ne sait pas si la durée voulue s'est écoulée ou non. Cette fonction est implémentée dans le bibliothèque GlibC en utilisant l'appel-système select(), que nous verrons dans le chapitre consacré aux traitements asynchrones.

Voici une variante du programme précédent, utilisant usleep(). Nous allons montrer que cette fonction peut aussi être interrompue par l'arrivée d'un signal.

exemple_usleep.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <unistd.h>
    #include <sys/wait.h>

    void
gestionnaire_sigusr1(int numero)
{
}

    int
main (void)
{
    pid_t            pid;
    struct sigaction action;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erreur dans fork \n");
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire_sigusr1;
    sigemptyset(& action.sa_mask);
    action.sa_flags = SA_RESTART;

    if (sigaction(SIGUSR1, & action, NULL) != 0) {
        fprintf(stderr, "Erreur dans sigaction \n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        system("date +\"%H:%M:%S\"");
        usleep(10000000); /* 10 millions de µs = 10 secondes */
        system("date +\"%H:%M:%S\"");
    } else {
        usleep(2000000); /* 2 millions de µs = 2 secondes */
        kill(pid, SIGUSR1);
        waitpid(pid, NULL, 0);
```

```
    }
    return EXIT_SUCCESS;
}
```

Le sommeil du processus fils, censé durer 10 secondes, est interrompu au bout de 2 secondes par le signal provenant du père, et ce malgré l'option `SA_RESTART` de `sigaction()`, comme le montre l'exécution suivante :

```
$ ./exemple_usleep
08:42:34
08:42:36
$
```

La fonction `usleep()` étant implémentée à partir de l'appel-système `select()`, elle n'a pas d'interaction inattendue avec un éventuel gestionnaire de signaux (sauf si ce dernier se termine par un saut `siglongjmp()` bien entendu).

Il existe encore une autre fonction de sommeil, offrant une précision encore plus grande : **`nanosleep()`**. Cette fonction est définie par SUSv3. Elle est déclarée ainsi dans `<time.h>` :

```
int nanosleep (const struct timespec * voulu, struct timespec * restant);
```

Le premier argument représente la durée de sommeil désirée, et le second argument, s'il est non `NULL`, permet de stocker la durée de sommeil restante lorsque la fonction a été interrompue par l'arrivée d'un signal. Si `nanosleep()` dort pendant la durée désirée, elle renvoie 0. En cas d'erreur, ou si un signal la réveille prématurément, elle renvoie –1 (et place `EINTR` dans `errno` dans ce dernier cas).

La structure `timespec` servant à indiquer la durée de sommeil a été décrite dans le chapitre précédent, avec l'appel-système `sigwaitinfo()`. Elle contient deux membres : l'un précisant le nombre de secondes, l'autre contenant la partie fractionnaire exprimée en nanosecondes.

Il est illusoire d'imaginer avoir une précision de l'ordre de la nanoseconde, ou même de la microseconde, sur un système multitâche. Même sur un système monotâche dédié à une application en temps-réel, il est difficile d'obtenir une telle précision sans avoir recours à des boucles d'attente vides, ne serait-ce qu'en raison de l'allongement de durée dû à l'appel-système proprement dit.

Quoi qu'il en soit, l'ordonnancement est soumis au séquencement de l'horloge interne, dont les intervalles sont séparés de 1/`HZ` seconde. La constante `HZ` est définie dans `<asm/param.h>`. Elle varie suivant les machines et les versions du noyau : depuis Linux 2.6, elle vaut 1000 sur les architectures x86 alors qu'auparavant elle valait 100. Ainsi un sommeil de processus est toujours arrondi à la milliseconde supérieure (au centième de seconde supérieur pour les noyaux 2.4 et précédents).

Bien que la précision de cette fonction soit illusoire, elle présente quand même un gros avantage par rapport aux deux précédentes. La fonction `usleep()` ne renvoyait pas la durée de sommeil restante en cas d'interruption, `sleep()`, nous l'avons vu, arrondissait cette valeur à la seconde supérieure, mais la pseudo-précision de `nanosleep()` permet de reprendre le sommeil interrompu en la rappelant directement avec la valeur de sommeil qui restait. Le calibrage à 1/`HZ` seconde (soit 1 millième de seconde sur les machines x86) permet de conserver une relative précision, même en cas de signaux fréquents. Dans le programme suivant, on va effectuer un sommeil de 60 secondes pendant lequel, le processus recevra chaque seconde cinq signaux (disons plutôt qu'il en recevra plusieurs, car certains seront certainement

regroupés). À chaque interruption, `nanosleep()` est relancée avec la durée restante, avec une pseudo précision à la nanoseconde.

exemple_nanosleep.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <time.h>
    #include <unistd.h>
    #include <sys/wait.h>

    void
gestionnaire_sigusr1(int numero)
{
}
    int
main (void)
{
    pid_t            pid;
    struct sigaction action;
    struct timespec  spec;
    int              i;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erreur dans fork \n");
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire_sigusr1;
    sigemptyset (& (action.sa_mask));
    action.sa_flags = SA_RESTART;

    if (sigaction(SIGUSR1, & action, NULL) != 0) {
        fprintf(stderr, "Erreur dans sigaction \n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        spec.tv_sec = 60;
        spec.tv_nsec = 0;
        system("date +\"%H:%M:%S\"");
        while (nanosleep(& spec, & spec) != 0)
            ;
        system("date +\"%H:%M:%S\"");
    } else {
        sleep(2); /* Pour éviter d'envoyer un signal pendant */
                  /* l'appel system() à /bin/date            */
        for (i = 0 ; i < 59; i ++) {
            sleep(1);
            kill(pid, SIGUSR1);
            kill(pid, SIGUSR1);
            kill(pid, SIGUSR1);
            kill(pid, SIGUSR1);
            kill(pid, SIGUSR1);
```

```
        }
        waitpid(pid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

L'exécution suivante nous montre qu'à un niveau macroscopique (la seconde), la précision est conservée, même sur une durée relativement longue comme une minute, avec une charge système assez faible.

```
$ ./exemple_nanosleep
13:04:05
13:05:05
$
```

Bien sûr dans notre cas, le gestionnaire de signaux n'effectuait aucun travail. Si le gestionnaire consomme vraiment du temps processeur, et si la précision du délai est critique, on se reportera au principe évoqué avec `sleep()`, en recadrant la durée restante régulièrement grâce à la fonction `time()`.

Dans les noyaux Linux 2.2 et 2.4, les attentes de faible durée (inférieures à 2 ms) avec `nanosleep()` pour les processus ordonnancés en temps-réel étaient réalisées en utilisant une boucle active. La précision était très bonne mais l'ensemble du système était gelé durant l'attente. Ce mécanisme a été abandonné depuis le noyau 2.6.

Sommeil utilisant les temporisations de précision

Nous avons vu dans les chapitres précédents le fonctionnement de l'appel-système `alarm()`, qui déclenche un signal `SIGALRM` au bout du nombre de secondes programmées. Il existe en fait trois temporisations qui fonctionnent sur un principe similaire, mais avec une plus grande précision (tout en étant toujours limitées à la résolution de l'horloge interne à 1/`HZ` seconde, soit 1 ms sur architecture PC).

De plus, ces temporisations peuvent être configurées pour redémarrer automatiquement au bout du délai prévu. Les trois temporisations sont programmées par l'appel-système `setitimer()`. On peut également consulter la programmation en cours grâce à l'appel `getitimer()`.

Le prototype de **`setitimer()`** est déclaré ainsi dans `<sys/time.h>` :

```
int setitimer (int laquelle, const struct itimerval * valeur,
               struct itimerval * ancienne);
```

Le premier argument permet de choisir quelle temporisation est utilisée parmi les trois constantes symboliques suivantes :

Nom	Signification
`ITIMER_REAL`	Le décompte de la temporisation a lieu en temps « réel », et lorsque le compteur arrive à zéro, le signal `SIGALRM` est envoyé au processus.
`ITIMER_VIRTUAL`	La temporisation ne décroît que lorsque le processus s'exécute en mode utilisateur. Un signal `SIGVTALRM` lui est envoyé à la fin du décompte.
`ITIMER_PROF`	Le décompte a lieu quand le processus s'exécute en mode utilisateur, mais également pendant qu'il s'exécute en mode noyau, durant les appels-système. Au bout du délai programmé, le signal `SIGPROF` est émis.

L'utilisation de la temporisation `ITIMER_REAL` est la plus courante. Elle s'apparente globalement au même genre d'utilisation que la fonction `alarm()`, mais offre une plus grande précision et un redémarrage automatique en fin de comptage.

`ITIMER_VIRTUAL` s'utilise surtout conjointement à `ITIMER_PROF`, car ces temporisations permettent, par une simple soustraction, d'obtenir des statistiques sur le temps d'exécution passé par le processus en mode utilisateur et en mode noyau.

La temporisation `ITIMER_PROF` permet de rendre compte du déroulement du processus indépendamment des mécanismes d'ordonnancement, et donc d'avoir une indication quantitative de la durée d'une tâche quelle que soit la charge système. On peut utiliser cette technique pour comparer par exemple les durées de plusieurs algorithmes de calcul.

Pour lire l'état de la programmation en cours, on utilise **`getitimer()`** :

```
int getitimer (int laquelle, struct itimerval * valeur);
```

La structure `itimerval` servant à stocker les données concernant un timer est définie dans `<sys/time.h>` avec les deux membres suivants :

Type	Nom	Signification
`struct timeval`	`it_interval`	Valeur à reprogrammer lors de l'expiration du timer
`struct timeval`	`it_value`	Valeur décroissante actuelle

La structure `timeval` que nous avons déjà rencontrée dans la présentation de `wait3()` est utilisée pour enregistrer les durées, avec les membres suivants :

Type	Nom	Signification
`time_t`	`tv_sec`	Nombre de secondes
`time_t`	`tv_usec`	Nombre de microsecondes

La valeur du membre `it_value` est décrémentée régulièrement suivant les caractéristiques de la temporisation. Lorsque cette valeur atteint zéro, le signal correspondant est envoyé. Puis, si la valeur du membre `it_interval` est non nulle, elle est copiée dans le membre `it_value`, et la temporisation repart.

La bibliothèque GlibC offre quelques fonctions d'assistance pour manipuler les structures `timeval`. Comme le champ `tv_usec` d'une telle structure doit toujours être compris entre 0 et 999.999, il n'est pas facile d'ajouter ou de soustraire ces données. Les fonctions d'aide sont les suivantes :

```
void timerclear (struct timeval * temporisation);
```

qui met à zéro les deux champs de la structure transmise.

```
void timeradd (const struct timeval * duree_1,
               const struct timeval * duree_2,
               struct timeval * duree_resultat);
```

additionne les deux structures (en s'assurant que les membres `tv_usec` ne dépassent pas 999.999) et remplit les champs de la structure résultat, sur laquelle on passe un pointeur en

dernier argument. Une structure utilisée en premier ou second argument peut aussi servir pour récupérer le résultat, la bibliothèque C réalisant correctement la copie des données.

```
void timersub (const struct timeval * duree_1,
               const struct timeval * duree_2,
               struct timeval * duree_resultat);
```

soustrait la deuxième structure de la première (en s'assurant que les membres `tv_usec` ne deviennent pas négatifs) et remplit les champs de la structure résultat.

```
int timerisset (const struct timeval * temporisation);
```

est vraie si au moins l'un des deux membres de la structure est non nul.

Attention

Nous avons présenté ici des prototypes de fonctions, mais en réalité elles sont toutes les quatre implémentées sous forme de macros, qui évaluent plusieurs fois leurs arguments. Il faut donc prendre les précautions adéquates pour éviter les effets de bord.

Le premier exemple que nous allons présenter avec `setitimer()` va servir à implémenter un sommeil de durée précise, même lorsque le processus reçoit de nombreuses interruptions parallèlement à son sommeil. Pour cela, nous utiliserons le timer `ITIMER_REAL`. Nous allons créer une fonction `sommeil_precis()`, prenant en argument le nombre de secondes, suivi du nombre de microsecondes de sommeil voulu. La routine sauvegarde tous les anciens paramètres, qu'elle modifie pour les rétablir en sortant. Elle renvoie 0 si elle réussit, ou –1 sinon. On utilise la méthode de blocage des signaux employant `sigsuspend()`, que nous avons étudié dans le chapitre précédent.

La routine `sommeil_precis()` créée ici est appelée depuis les deux processus père et fils que nous déclenchons dans `main()`. Le fils utilise un appel sur une longue durée (60 s), et le père une multitude d'appels de courte durée (20 ms). Le processus père envoie un signal `SIGUSR1` à son fils entre chaque petit sommeil.

Les deux processus invoquent la commande `date` au début et à la fin de leur exécution pour afficher l'heure.

exemple_setitimer_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <errno.h>
    #include <sys/time.h>
    #include <sys/wait.h>

    static int      temporisation_ecoulee;

    void
gestionnaire_sigalrm (int inutile)
{
    temporisation_ecoulee = 1;
}
```

```
    int
sommeil_precis (long nb_secondes, long nb_microsecondes)
{
    struct sigaction action;
    struct sigaction ancienne_action;
    sigset_t         masque_sigalrm;
    sigset_t         ancien_masque;
    int              sigalrm_dans_ancien_masque = 0;
    struct itimerval ancien_timer;
    struct itimerval nouveau_timer;
    int              retour = 0;

    /* Préparation du timer */
    timerclear(& (nouveau_timer . it_interval));
    nouveau_timer.it_value.tv_sec = nb_secondes;
    nouveau_timer.it_value.tv_usec = nb_microsecondes;

    /* Installation du gestionnaire d'alarme */
    action.sa_handler = gestionnaire_sigalrm;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = SA_RESTART;
    if (sigaction(SIGALRM, & action, & ancienne_action) != 0)
        return -1;

    /* Blocage de SIGALRM avec mémorisation du masque en cours */
    sigemptyset(& masque_sigalrm);
    sigaddset(& masque_sigalrm, SIGALRM);
    if (sigprocmask(SIG_BLOCK, & masque_sigalrm, & ancien_masque) != 0) {
        retour = -1;
        goto reinstallation_ancien_gestionnaire;
    }
    if (sigismember(& ancien_masque, SIGALRM)) {
        sigalrm_dans_ancien_masque = 1;
        sigdelset(& ancien_masque, SIGALRM);
    }
    /* Initialisation de la variable globale */
    temporisation_ecoulee = 0;

    /* Sauvegarde de l'ancien timer */
    if (getitimer(ITIMER_REAL, & ancien_timer) != 0) {
        retour = -1;
        goto restitution_ancien_masque;
    }
    /* Déclenchement du nouveau timer */
    if (setitimer(ITIMER_REAL, & nouveau_timer, NULL) != 0) {
        retour = -1;
        goto restitution_ancien_timer;
    }
    /* Boucle d'attente de la fin du sommeil */
    while (! temporisation_ecoulee) {
        if ((sigsuspend(& ancien_masque) != 0) &&
            (errno != EINTR)) {
```

```
            retour = -1;
            break;
        }
    }
restitution_ancien_timer:
    if (setitimer(ITIMER_REAL, & ancien_timer, NULL) != 0)
        retour = -1;
restitution_ancien_masque :
    if (sigalrm_dans_ancien_masque) {
        sigaddset(& ancien_masque, SIGALRM);
    }
    if (sigprocmask(SIG_SETMASK, & ancien_masque, NULL) != 0)
        retour = -1;
reinstallation_ancien_gestionnaire :
    if (sigaction(SIGALRM, & ancienne_action, NULL) != 0)
        retour = -1;

    return retour;
}

    void
gestionnaire_sigusr1 (int inutile)
{
}

    int
main (void)
{
    pid_t            pid;
    struct sigaction action;
    int              i;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erreur dans fork \n");
        exit(EXIT_FAILURE);
    }
    action.sa_handler = gestionnaire_sigusr1;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = SA_RESTART;

    if (sigaction(SIGUSR1, & action, NULL) != 0) {
        fprintf(stderr, "Erreur dans sigaction \n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        system("date +\"Fils : %H:%M:%S\"");
        if (sommeil_precis(60, 0) != 0) {
            fprintf(stderr, "Erreur dans sommeil_precis \n");
            exit(EXIT_FAILURE);
        }
        system("date +\"Fils : %H:%M:%S\"");
    } else {
```

```
        sommeil_precis(2, 0);
        system("date +\"Père : %H:%M:%S\"");
        for (i = 0; i < 3000; i ++) {
            sommeil_precis(0, 20000); /* 1/50 de seconde */
            kill(pid, SIGUSR1);
        }
        system("date +\"Père : %H:%M:%S\"");
        waitpid(pid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

Nous voyons que la précision du sommeil est bien conservée, tant sur une longue période que sur un court intervalle de sommeil :

```
$ ./exemple_setitimer_1
Fils : 17:50:34
Père : 17:50:36
Fils : 17:51:34
Père : 17:51:36
$
```

Les temporisations n'expirent jamais avant la fin du délai programmé, mais plutôt légèrement après, avec un retard constant dépendant de l'horloge interne du système. Si on désire faire des mesures critiques, il est possible de calibrer ce léger retard.

Avec la temporisation `ITIMER_REAL`, lorsque le signal `SIGALRM` est émis, le processus n'est pas nécessairement actif (contrairement aux deux autres temporisations). Il peut donc s'écouler un retard avant l'activation du processus et la délivrance du signal. Avec la temporisation `ITIMER_PROF`, le processus peut se trouver au sein d'un appel-système, et un retard sera également possible avant l'appel du gestionnaire de signaux.

Notre second exemple va utiliser conjointement les deux timers `ITIMER_VIRTUAL` et `ITIMER_PROF` pour mesurer les durées passées dans les modes utilisateur et noyau d'une routine qui fait une série de boucles consommant du temps processeur, suivie d'une série de copies d'un fichier vers le périphérique `/dev/null` pour exécuter de nombreux appels-système.

Le gestionnaire de signaux commun aux deux temporisations départage les signaux, puis incrémente le compteur correspondant. Les temporisations sont réglées pour envoyer un signal tous les centièmes de seconde. Une routine d'affichage des données est installée par `atexit()` afin d'être invoquée en sortie du programme.

exemple_setitimer_2.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/wait.h>

unsigned long int       mode_utilisateur;
unsigned long int       mode_utilisateur_et_noyau;
```

```
    void    gestionnaire_signaux    (int numero);
    void    fin_du_suivi            (void);
    void    action_a_mesurer        (void);

    int
main (void)
{
    struct sigaction action;
    struct itimerval timer;

    /* Préparation du timer */
    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = 10000; /* 1/100 s. */
    timer.it_interval.tv_sec = 0;
    timer.it_interval.tv_usec = 10000; /* 1/100 s. */

    /* Installation du gestionnaire de signaux */
    action.sa_handler = gestionnaire_signaux;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = SA_RESTART;
    if ((sigaction(SIGVTALRM, & action, NULL) != 0)
     || (sigaction(SIGPROF, & action, NULL) != 0)) {
        fprintf(stderr, "Erreur dans sigaction \n");
        return EXIT_FAILURE;
    }
    /* Déclenchement des nouveaux timers */
    if ((setitimer(ITIMER_VIRTUAL, & timer, NULL) != 0)
     || (setitimer(ITIMER_PROF, & timer, NULL) != 0)) {
        fprintf(stderr, "Erreur dans setitimer \n");
        return EXIT_FAILURE;
    }
    /* Installation de la routine de sortie du programme */
    if (atexit(fin_du_suivi) != 0) {
        fprintf(stderr, "Erreur dans atexit \n");
        return EXIT_FAILURE;
    }
    /* Appel de la routine de travail effectif du processus */
    action_a_mesurer();
    return EXIT_SUCCESS;
}

    void
gestionnaire_signaux (int numero)
{
    switch (numero) {
        case SIGVTALRM :
            mode_utilisateur ++;
            break;
        case SIGPROF :
            mode_utilisateur_et_noyau ++;
            break;
    }
```

```
}

    void
fin_du_suivi (void)
{
    sigset_t        masque;

    /* Blocage des signaux pour éviter une modification */
    /* des compteurs en cours de lecture.               */
    sigemptyset(& masque);
    sigaddset(& masque, SIGVTALRM);
    sigaddset(& masque, SIGPROF);
    sigprocmask(SIG_BLOCK, & masque, NULL);

    /* Comme on quitte à présent le programme, on ne
     * restaure pas l'ancien comportement des timers,
     * mais il faudrait le faire dans une routine de
     * bibliothèque.
     */
    fprintf(stdout, "Temps passé en mode utilisateur : %ld/100 s \n",
                mode_utilisateur);
    fprintf(stdout, "Temps passé en mode noyau : %ld/100 s \n",
                mode_utilisateur_et_noyau - mode_utilisateur);
}

    void
action_a_mesurer (void)
{
    int i, j;
    FILE * fp1, * fp2;
    double x;

    x = 0.0;
    for (i = 0; i < 20000; i ++)
        for (j = 0; j < 20000; j ++)
            x += i * j;
    for (i = 0; i < 5000; i ++) {
        if ((fp1 = fopen ("exemple_setitimer_2", "r")) != NULL) {
            if ((fp2 = fopen("/dev/null", "w")) != NULL) {
                while (fread(& j, sizeof (int), 1, fp1) == 1)
                    fwrite(& j, sizeof (int), 1, fp2);
                fclose(fp2);
            }
            fclose(fp1);
        }
    }
}
```

L'exécution affiche les résultats suivants :

```
$ ./exemple_setitimer_2
Temps passé en mode utilisateur : 629/100 s
Temps passé en mode noyau : 14/100 s
```

```
$ ./exemple_setitimer_2
Temps passé en mode utilisateur : 631/100 s
Temps passé en mode noyau : 13/100 s
$ ./exemple_setitimer_2
Temps passé en mode utilisateur : 620/100 s
Temps passé en mode noyau : 13/100 s
$
```

Nous voyons les limites du suivi d'exécution sur un système multitâche, même si les ordres de grandeur restent bien constants. Nous copions à présent ce programme dans `exemple_setitimer_3.c` en ne conservant plus que la routine de travail effectif, ce qui nous donne cette fonction `main()` :

```
    int
main (void)
{
    action_a_mesurer();
    return EXIT_SUCCESS;
}
```

Nous pouvons alors utiliser la fonction « `times` » de `bash` 2, qui permet de mesurer les temps cumulés d'exécution en mode noyau et en mode utilisateur du shell et des processus qu'il a lancés.

```
$ sh -c "./exemple_setitimer_3 ; times"
0m0.000s 0m0.002s
0m6.545s 0m0.169s
$ sh -c "./exemple_setitimer_3 ; times"
0m0.000s 0m0.002s
0m6.563s 0m0.134s
$ sh -c "./exemple_setitimer_3 ; times"
0m0.000s 0m0.0002s
0m6.562s 0m0.149s
$
```

Nous voyons que les résultats sont tout à fait comparables, même s'ils présentent également une variabilité due à l'ordonnancement multitâche.

Timers temps-réel

Depuis le noyau 2.6, Linux intègre un ensemble d'appels système permettant de réaliser des tâches périodiques. Le principe est simple : on programme un *timer* avec un délai initial et une période de répétition donnés. Au déclenchement du timer, le noyau délivrera un signal au processus et réarmera la temporisation. La création et la destruction d'un timer se font avec les routines suivantes :

```
int timer_create (clockid_t clock, struct sigevent * event,
                  timer_t * timer);
int timer_delete (timer_t timer);
```

Le premier argument de `timer_create()`, de type `clockid_t`, décrit une horloge fournie par le système sur laquelle le timer doit s'appuyer. Chaque implémentation peut proposer ses propres horloges, et éventuellement proposer la définition d'horloges par l'application.

Toutefois certaines ont une signification standard :

- `CLOCK_REALTIME` : cette horloge représente l'écoulement effectif du temps tel qu'il est perçu par le système. La valeur de l'horloge peut être modifiée (par exemple avec la commande `date` depuis le shell).
- `CLOCK_MONOTONIC` : contrairement à la précédente, la valeur de l'horloge monotone est strictement croissante. La modification de l'heure système n'influe pas sur elle.
- `CLOCK_PROCESS_CPUTIME_ID` : une horloge qui démarre en même temps que le processus. Elle représente la durée complète d'exécution du processus, même lorsqu'il est endormi.
- `CLOCK_THREAD_CPUTIME_ID` : cette horloge est semblable à la précédente mais démarre lorsque le *thread* courant débute. Elle peut donc être décalée par rapport à la précédente.

Le second argument est une structure décrivant la manière dont le noyau doit notifier le processus de l'expiration du timer. Cette structure sera étudiée plus en détail dans le chapitre 30, en examinant les fonctions `aio_read()` et `aio_write()`. Ici nous nous limiterons à son champ `sigev_notify` qui doit contenir la constante `SIGEV_SIGNAL` pour que le noyau sache qu'il doit envoyer un signal au processus. En outre, le champ `sigev_signo` contient le numéro du signal à employer.

Le troisième argument enfin est un pointeur sur un objet de type `timer_t` qui contiendra au retour l'identifiant du timer créé.

Pour amorcer le timer, il faut utiliser la fonction suivante :

```
int timer_settime (timer_t timer, int options,
                   const struct itimerspec * spec,
                   struct itimerspec * precedent);
```

Le premier argument est l'identifiant du timer obtenu précédemment, si le second argument est nul, les valeurs indiquées dans la structure `itimerspec` du troisième argument sont relatives à l'heure actuelle. Si ce second argument contient la valeur `TIMER_ABSTIME`, les valeurs sont des heures absolues.

La structure `itimerspec` est très proche de la structure `itimerval` que nous avons vue plus haut. Elle contient deux champs `it_value` et `it_interval` de type `struct timespec` (deux champs `tv_sec` et `tv_nsec`) décrivant (en secondes et nanosecondes) le délai avant expiration du timer et la période de répétition.

Ces fonctions de temporisation étant issues de la norme Posix.1b (temps-réel), il faut ajouter l'option `–lrt` (*real time library*) lors de l'édition des liens. Ceci est présent dans le fichier `Makefile` fourni avec les sources des programmes du livre.

Dans l'exemple suivant nous allons programmer deux timers : le premier s'exécute avec une fréquence fournie par l'utilisateur sur la ligne de commande. À chaque expiration nous incrémentons un compteur global. Le second timer fonctionne avec une période d'une seconde. Il affiche la valeur du compteur (qui correspond donc à la fréquence réelle de l'autre timer) et le réinitialise.

exemple_timer.c :

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
        #include <unistd.h>

        volatile int compteur = 0;

        void
gestionnaire_usr1(int numero)
{
        compteur ++;
}
        void
gestionnaire_usr2(int numero)
{
        fprintf(stderr, "%d\n", compteur);
        compteur = 0;
}

        int
main(int argc, char * argv[])
{
        long int        frequence;

        timer_t timer1, timer2;
        struct sigevent event;
        struct itimerspec itimer;

        struct sigaction action;

        if ((argc != 2)
         || (sscanf(argv[1], "%ld", & frequence) != 1)
         || (frequence < 2) || (frequence > 1000000000)) {
                fprintf(stderr, "usage: %s frequence\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        action.sa_flags = 0;
        sigfillset(& action.sa_mask);

        action.sa_handler = gestionnaire_usr1;
        sigaction(SIGUSR1, & action, NULL);

        action.sa_handler = gestionnaire_usr2;
        sigaction(SIGUSR2, & action, NULL);

        event.sigev_notify = SIGEV_SIGNAL;
        event.sigev_signo  = SIGUSR1;

        if (timer_create(CLOCK_REALTIME, & event, & timer1) != 0) {
                perror("timer_create");
                exit(EXIT_FAILURE);
        }
```

```
        event.sigev_notify = SIGEV_SIGNAL;
        event.sigev_signo  = SIGUSR2;

        if (timer_create(CLOCK_REALTIME, & event, & timer2) != 0) {
                perror("timer_create");
                exit(EXIT_FAILURE);
        }

        itimer.it_value.tv_sec = 0;
        itimer.it_value.tv_nsec = 1000000000/frequence;
        itimer.it_interval.tv_sec = 0;
        itimer.it_interval.tv_nsec = 1000000000/frequence;

        if (timer_settime(timer1, 0, & itimer, NULL) != 0) {
                perror("timer_settime");
                exit(EXIT_FAILURE);
        }

        itimer.it_value.tv_sec = 1;
        itimer.it_value.tv_nsec = 0;
        itimer.it_interval.tv_sec = 1;
        itimer.it_interval.tv_nsec = 0;

        if (timer_settime(timer2, 0, & itimer, NULL) != 0) {
                perror("timer_settime");
                exit(EXIT_FAILURE);
        }

        while (1)
                pause();
        return EXIT_SUCCESS;
}
```

Lors de son exécution, le programme montre les limites d'un timer géré par l'ordonnanceur. Pour de faibles valeurs, le comptage est assez bon, mais très vite la limitation à la milliseconde supérieure est contraignante :

```
$ ./exemple_timer 10
9
10
10
10
  (Ctrl-C)
$ ./exemple_timer 20
19
20
19
20
  (Ctrl-C)
$ ./exemple_timer 50
47
48
```

```
48
47
  (Ctrl-C)
$ ./exemple_timer 100
91
91
91
  (Ctrl-C)
$ ./exemple_timer 200
166
167
  (Ctrl-C)
$ ./exemple_timer 500
333
334
  (Ctrl-C)
$ ./exemple_timer 501
501
500
  (Ctrl-C)
$ ./exemple_timer 1000
500
501
  (Ctrl-C)
$ ./exemple_timer 1001
1001
1001
  (Ctrl-C)
$ ./exemple_timer 2000
1001
1001
  (Ctrl-C)
$ ./exemple_timer 10000
1001
1001
  (Ctrl-C)
$
```

Le noyau Linux standard n'est pas fait pour réaliser des tâches temps-réel de haute précision. Si l'on a besoin d'obtenir des temporisations de durée inférieure à quelques millisecondes, il faudra utiliser une autre solution.

Une alternative existe : le projet RTAI (hébergé par l'université polytechnique de Milan sur le site : *http://www.aero.polimi.it/~rtai*) fait fonctionner le noyau Linux comme tâche de fond d'un micro-noyau minimal offrant des fonctionnalités temps-réel strictes. Il est possible de programmer des tâches pour le noyau minimal (par exemple des acquisitions de données avec une périodicité très précise) qui communiqueront avec l'application principale tournant dans l'espace du noyau Linux.

D'autres implémentations du même type existent, mais RTAI est l'option la plus avancée développée sous licence libre.

Suivre l'exécution d'un processus

Il existe plusieurs fonctions permettant de suivre l'exécution d'un processus, à la manière des routines que nous avons développées précédemment. La plus simple d'entre elles est la fonction **clock()**, déclarée dans <time.h> ainsi :

```
clock_t clock (void);
```

Le type clock_t représente un temps processeur écoulé sous forme d'impulsions d'horloge théoriques. Nous précisons qu'il s'agit d'impulsions *théoriques* car il y a une différence d'ordre de grandeur importante entre ces quantités et la véritable horloge système utilisée par l'ordonnanceur. À cause d'une différence entre les standards Ansi C et Posix, cette valeur n'a plus aucune signification effective. Pour obtenir une durée en secondes, il faut diviser la valeur clock_t par la constante CLOCKS_PER_SEC. Cette constante vaut 1 million sur l'essentiel des systèmes Unix dérivant de Système V, ainsi que sous Linux. On imagine assez bien que le séquencement des tâches est loin d'avoir effectivement lieu toutes les microsecondes…

On ne sait pas avec quelle valeur la fonction clock() démarre. Il s'agit parfois de zéro, mais ce n'est pas obligatoire. Aussi est-il nécessaire de mémoriser la valeur initiale et de la soustraire pour connaître la durée écoulée.

Sous Linux, clock_t est un entier long, mais ce n'est pas toujours le cas sur d'autres systèmes. Il importe donc de forcer le passage en virgule flottante pour pouvoir effectuer l'affichage. Notre programme d'exemple va mesurer le temps processeur écoulé tant en mode utilisateur qu'en mode noyau, dans la routine que nous avons déjà utilisée dans les exemples précédents.

Le programme exemple_clock.c contient donc la fonction main() suivante, en plus de la routine action_a_mesurer().

exemple_clock.c :

```
    int
main (void)
{
    clock_t debut_programme;
    double  duree_ecoulee;

    debut_programme = clock();

    action_a_mesurer();

    duree_ecoulee = clock() - debut_programme;
    duree_ecoulee = duree_ecoulee / CLOCKS_PER_SEC;
    fprintf(stdout, "Durée = %f \n", duree_ecoulee);
    return EXIT_SUCCESS;
}
```

Les résultats sont les suivants :

```
$ ./exemple_clock
Durée = 6.800000
$ ./exemple_clock
Durée = 6.780000
$
```

Comme il fallait s'y attendre, il s'agit de la somme des temps obtenus avec nos programmes précédents, avec – comme toujours – une légère variation en fonction de la charge système.

Sous Linux, `clock_t` est équivalent à un `long int`. Il y a donc un risque de débordement de la valeur maximale au bout de `LONG_MAX` impulsions théoriques. `LONG_MAX` est une constante symbolique définie dans `<limits.h>`. Sur un PC, `LONG_MAX` vaut 2.147.483.647, et `CLOCKS_PER_SEC` vaut 1.000.000. Cela donne donc une durée avant le dépassement de 2 147 secondes, soit 35 minutes.

Rappelons qu'il s'agit là de temps processeur effectif, et qu'il est assez rare qu'un programme cumule autant de temps d'exécution. Mais cela peut arriver, notamment avec des programmes de calcul ou de traitement d'image. Si on désire suivre les durées d'exécution de tels programmes (particulièrement pour comparer des algorithmes), il faut disposer d'un mécanisme permettant d'obtenir des mesures plus longues.

L'appel-système **`times()`** fournit ces informations. Il est déclaré ainsi dans `<sys/times.h>` :

```
clock_t times (struct tms * mesure);
```

La valeur renvoyée par cet appel-système est le nombre de **jiffies**, c'est-à-dire le nombre de cycles d'horloge exécutés depuis le démarrage du système, ou `(clock_t)-1` en cas d'échec. Il s'agit cette fois de la véritable horloge de l'ordonnanceur, qui a une période de 1/1000 s sur PC. On peut utiliser cette valeur renvoyée, un peu à la manière de celle qui est fournie par la fonction `clock()`, mais en utilisant une autre constante de conversion, celle qui décrit le nombre d'impulsions d'horloge par seconde : `CLK_TCK`, définie dans `<time.h>`.

Notre premier exemple ne s'occupera pas de l'argument de la fonction `times()`, en passant un pointeur `NULL`, afin de se soucier uniquement de la valeur de retour. Nous calquons notre programme sur le précédent, avec la fonction `main()` suivante :

exemple_times_1.c :

```
    int
main (void)
{
    clock_t debut_programme;
    double  duree_ecoulee;

    debut_programme = times (NULL);
    fprintf(stdout, "Jiffies au début %ld \n", debut_programme);

    action_a_mesurer();

    fprintf(stdout, "Jiffies en fin %ld \n", times (NULL));
    duree_ecoulee = times(NULL) - debut_programme;
    duree_ecoulee = duree_ecoulee / CLK_TCK;
    fprintf(stdout, "Durée = %f \n", duree_ecoulee);
    return EXIT_SUCCESS;
}
```

L'exécution est la suivante :

```
$ ./exemple_times_1
Jiffies au début 444792126
```

```
Jiffies en fin 444792838
Durée = 7.120000
$
```

Cette fois-ci, la durée est celle de l'exécution totale du programme et non le temps processeur consommé (il y a peu d'écart car notre machine est actuellement faiblement chargée).

Voici à présent le détail de la structure **`tms`**, qu'on passe en argument de `times()` afin qu'elle soit remplie. La définition se trouve dans `<sys/times.h>`.

Type	Nom	Signification
`clock_t`	`tms_utime`	Temps processeur passé en mode utilisateur
`clock_t`	`tms_stime`	Temps processeur passé en mode noyau
`clock_t`	`tms_cutime`	Temps processeur passé en mode utilisateur par les processus fils terminés du programme appelant
`clock_t`	`tms_cstime`	Temps processeur passé en mode noyau par les processus fils terminés du programme appelant

Les deux derniers membres contiennent les temps utilisateur et noyau cumulés de tous les processus fils terminés au moment de l'appel. Nous allons utiliser ces membres après avoir invoqué une commande passée en argument au programme. Cela permet d'avoir une fonctionnalité du même style que la commande `times` intégrée au shell `bash`.

exemple_times_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <time.h>
    #include <sys/times.h>

    int
main (int argc, char * argv[])
{
    struct tms mesure;
    double duree_ecoulee;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s <commande> \n", argv [0]);
        exit(EXIT_FAILURE);
    }
    system(argv[1]);
    times(& mesure);
    duree_ecoulee = mesure.tms_cutime;
    duree_ecoulee = duree_ecoulee / CLK_TCK;
    fprintf(stdout, "Temps CPU mode utilisateur = %f \n", duree_ecoulee);
    duree_ecoulee = mesure.tms_cstime;
    duree_ecoulee = duree_ecoulee / CLK_TCK;
    fprintf(stdout, "Temps CPU en mode noyau    = %f \n", duree_ecoulee);
    return EXIT_SUCCESS;
}
```

L'exécution, en reprenant toujours notre même routine de test, donne les résultats suivants :

```
$ ./exemple_times_2 ./exemple_setitimer_3
Temps CPU mode utilisateur = 6.560000
Temps CPU en mode noyau    = 0.130000
$ ./exemple_times_2 ./exemple_setitimer_3
Temps CPU mode utilisateur = 6.550000
Temps CPU en mode noyau    = 0.150000
$ ./exemple_times_2 ./exemple_setitimer_3
Temps CPU mode utilisateur = 6.540000
Temps CPU en mode noyau    = 0.150000
$
```

Les statistiques obtenues sont compatibles avec les données que nous avons déjà observées.

Obtenir des statistiques sur un processus

Il est parfois utile d'obtenir des informations sur les performances d'un programme donné. On peut ainsi, dans certains cas, optimiser les algorithmes ou réétudier certains goulets d'étranglement où le programme est ralenti. La fonction `getrusage()` permet d'obtenir les statistiques concernant le processus appelant ou l'ensemble de ses fils qui se sont terminés. Dans ce dernier cas, les valeurs sont cumulées sur la totalité des processus concernés.

Le prototype de **`getrusage()`** est déclaré dans `<sys/resource.h>` ainsi :

```
int getrusage (int lesquelles, struct rusage * statistiques);
```

Le premier argument de cette fonction indique quelles statistiques nous intéressent. Il peut s'agir de l'une des constantes symboliques suivantes :

Nom	Signification
`RUSAGE_SELF`	Obtenir les informations concernant le processus appelant
`RUSAGE_CHILDREN`	Obtenir les statistiques sur les processus fils terminés

Le second argument est la structure `rusage`, que nous avons déjà rencontrée dans le paragraphe concernant `wait3()`. Cette dernière est remplie lors de l'appel. La fonction renvoie 0 si elle réussit, et –1 si elle échoue. En fait, sous Linux, comme avec `wait3()` et `wait4()`, l'appel-système `getrusage()` ne remplit qu'un petit nombre de champs de la structure `rusage`.

Type	Nom	Signification
`struct timeval`	`ru_utime`	Temps passé par le processus en mode utilisateur.
`struct timeval`	`ru_stime`	Temps passé par le processus en mode noyau.
`long`	`ru_minflt`	Nombre de fautes de pages mineures (n'ayant pas nécessité de rechargement depuis le disque).
`long`	`ru_majflt`	Nombre de fautes de pages majeures (ayant nécessité un rechargement des données depuis le disque).
`long`	`ru_nswap`	Nombre de fois où le processus a été entièrement swappé.

Les autres champs sont mis à zéro lors de l'appel. De cette façon, une application désirant tirer parti de leur contenu lorsqu'ils seront vraiment remplis par une nouvelle version du noyau peut vérifier si leur valeur est non nulle et les utiliser alors.

Voici un exemple d'utilisation où, comme dans le programme précédent, nous lançons la commande passée en argument grâce à la fonction `system()`. Si aucun argument n'est fourni, le processus affiche les statistiques le concernant.

exemple_rusage.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/resource.h>

    int
main (int argc, char * argv[])
{
    int lesquelles;
    struct rusage statistiques;

    if (argc == 1) {
        lesquelles = RUSAGE_SELF;
    } else {
        system (argv[1]);
        lesquelles = RUSAGE_CHILDREN;
    }
    if (getrusage(lesquelles, & statistiques) != 0) {
        fprintf(stderr, "Impossible d'obtenir les statistiques \n");
        exit(EXIT_FAILURE);
    }
    if (getrusage(lesquelles, & statistiques) != 0) {
        fprintf(stderr, "Impossible d'obtenir les statistiques \n");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Temps en mode utilisateur  %ld s. et %ld ms \n",
        statistiques.ru_utime.tv_sec,
        statistiques.ru_utime.tv_usec / 1000);
    fprintf(stdout, "Temps en mode noyau        %ld s. et %ld ms \n",
        statistiques.ru_stime.tv_sec,
        statistiques.ru_stime.tv_usec / 1000);
    fprintf(stdout, "\n");
    fprintf(stdout, "Nombre de fautes de pages mineures : %ld \n",
        statistiques.ru_minflt);
    fprintf(stdout, "Nombre de fautes de pages majeures : %ld \n",
        statistiques.ru_majflt);
    fprintf(stdout, "Nombre de swaps du processus       : %ld \n",
        statistiques.ru_nswap);
    return EXIT_SUCCESS;
}
```

On l'exécute toujours avec la même routine de test :

```
$ ./exemple_rusage ./exemple_setitimer_3
Temps en mode utilisateur  6 s. et 549 ms
Temps en mode noyau        0 s. et 141 ms

Nombre de fautes de pages mineures : 10374
Nombre de fautes de pages majeures : 0
Nombre de swaps du processus       : 0
$
```

Les durées sont cohérentes avec les statistiques déjà obtenues. Le nombre de fautes de pages est beaucoup plus difficile à interpréter.

Limiter les ressources consommées par un processus

Les processus disposent d'un certain nombre de limites, supervisées par le noyau. Celles-ci se rapportent à des paramètres concernant des ressources système dont l'utilisation par le processus est surveillée étroitement. Certaines limites se justifient principalement dans le cas d'un système multi-utilisateur, pour éviter de léser les autres personnes connectées. C'est le cas par exemple du nombre maximal de processus pour un même utilisateur, ou de la taille maximale d'un fichier. D'autres limites peuvent intéresser le programmeur, pour surveiller le comportement de son application, comme la limite maximale de la pile ou du temps processeur consommé.

Comme la plupart des stations Linux sont réservées à un seul utilisateur, une partie importante des limites ne se justifie pas forcément, et les distributions ont un comportement très libéral dans leur configuration par défaut.

On accède aux ressources d'un processus grâce à la fonction **`getrlimit()`**, déclarée dans `<sys/resource.h>` ainsi :

```
int getrlimit (int ressource, struct rlimit * limite);
```

Le premier argument permet de préciser la ressource concernée, et la structure `rlimit`, transmise en second argument, est remplie avec la limite demandée. La fonction renvoie 0 si elle réussit, et –1 en cas d'échec.

Chaque limite est composée de deux valeurs : une souple et une stricte. La limite souple peut être augmentée ou diminuée au gré de l'utilisateur, tout en ne dépassant jamais la limite stricte. Celle-ci peut être diminuée par l'utilisateur, mais ne peut être augmentée que par *root* ou un processus ayant la capacité `CAP_SYS_RESOURCE`.

Les limites sont transmises aux processus fils et aux programmes lancés par un `exec()`. Il est donc courant que l'administrateur système impose des valeurs aux limites strictes dans les fichiers d'initialisation du shell de connexion des utilisateurs.

Il existe une valeur spéciale, définie par la constante symbolique `RLIM_INFINITY`, pour indiquer que le processus n'a pas de limitation pour la ressource concernée.

Les différentes ressources sont les suivantes :

Nom	Signification
RLIMIT_CPU	Temps processeur consommable par le processus. S'il dépasse sa limite souple, il reçoit le signal SIGXCPU toutes les secondes. S'il l'ignore et atteint sa limite stricte, le noyau le tue par SIGKILL.
RLIMIT_FSIZE	Taille maximale en octets d'un fichier créé par le processus. Si on essaye de dépasser cette valeur, un signal SIGXFSZ est envoyé au processus si aucun octet n'a été écrit. Si ce signal est ignoré, les écritures après la limite se solderont par un échec avec l'erreur EFBIG.
RLIMIT_DATA	Taille maximale de la zone de données d'un processus. Cette limite est mise en place au chargement du programme.
RLIMIT_STACK	Taille maximale de la pile.
RLIMIT_CORE	Taille maximale d'un éventuel fichier d'image mémoire core. Si cette limite est mise à zéro, aucun fichier core ne sera créé en cas d'arrêt anormal du processus.
RLIMIT_RSS	Taille maximale de l'ensemble des données se trouvant simultanément en mémoire (cette limite n'est pas utilisée par les noyaux 2.2 et 2.6, et son usage sous Linux 2.4 est très limité).
RLIMIT_NPROC	Nombre maximal de processus simultanés pour l'utilisateur.
RLIMIT_NOFILE	Nombre maximal de fichiers ouverts simultanément par un utilisateur.
RLIMIT_MEMLOCK	Taille maximale de la zone verrouillée en mémoire centrale en empêchant son transfert dans le swap. On étudiera le détail de ce mécanisme avec les fonctions comme mlock().

Ces limites sont surtout utiles pour empêcher un utilisateur de s'approprier trop de ressources au détriment des autres. Elles n'ont pas un grand intérêt pour le programmeur, à quelques exceptions près :

- La limite de temps CPU : pour des applications effectuant de lourds calculs ou des logiciels fonctionnant sans interruption pendant plusieurs mois, il est bon de vérifier que la limite de temps CPU n'est pas trop restrictive. Sinon, il faudra demander à l'administrateur système une augmentation de la limite stricte.
- La limite de taille de fichier : en général, cette limite est suffisamment grande pour les applications courantes. Elle peut cependant être contraignante, par exemple pour un système d'enregistrement sur une longue durée de données provenant d'un réseau. Il peut alors être nécessaire de scinder un gros fichier en plusieurs petits. Notons que cette limitation ne fonctionne pas sur tous les systèmes de fichiers et que d'autres limites peuvent être imposées, comme le système de quotas de disques, ou des limites sur le serveur d'une partition montée par NFS. Lorsqu'un processus tente de dépasser cette taille, il reçoit un signal SIGXFSZ si aucun octet n'a pu être écrit.
- La limite de taille des fichiers core : lorsqu'un programme a terminé sa phase de débogage et qu'il est livré aux utilisateurs, les éventuels fichiers core qui peuvent être créés lorsqu'il s'arrête anormalement ne présentent aucun intérêt pour l'utilisateur final. Ils laissent même une impression de finition négligée s'ils ont tendance à se multiplier dans tous les répertoires où l'utilisateur se trouve lorsqu'il lance l'application. Il est donc conseillé, au début du programme, de mettre à zéro cette limite lorsqu'on décide que le code est suffisamment stable pour être distribué aux clients.

- Le nombre maximal de processus simultanés : il est conseillé de mettre une valeur suffisamment haute (par exemple, 256) dans les fichiers d'initialisation du shell de connexion, mais de ne pas laisser la limite infinie. En effet, cela permet de prévenir des erreurs de programmation où on boucle sur un `fork()`. Au bout d'un certain temps, celui-ci échouera et, de cette manière, *root* pourra stopper tous les processus du groupe fautif depuis une autre console.

Pour accéder aux limites, les shells offrent une commande intégrée, qui nous suffira dans la plupart des cas puisque les limites sont transmises aux processus fils, donc aux applications lancées par le shell.

Attention

Le shell n'utilise pas toujours les mêmes unités que celles manipulées par le noyau. Ainsi la limite `FSIZE` (taille maximale de fichier) est exprimée en kilo-octets par le shell et en octets par le noyau.

Avec *tcsh*, la commande est « `limit` ». Si on l'invoque seule, elle affiche l'état des limites souples actuelles. Avec l'option `-h`, elle s'occupe des limites strictes (hard).

```
% limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    1000000 kbytes
memoryuse       unlimited
descriptors     1024
memorylocked    unlimited
maxproc         256
openfiles       1024
% limit -h
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       unlimited
coredumpsize    unlimited
memoryuse       unlimited
descriptors     1024
memorylocked    unlimited
maxproc         256
openfiles       1024
%
```

Nous voyons une différence sur les limitations de taille de la pile et des fichiers `core`, qui doivent probablement être fixées dans un script d'initialisation de *tcsh*.

Si on veut modifier une limite, on indique le nom de la ressource, tel qu'il apparaît dans la liste précédente, suivi de la valeur désirée (en secondes pour le temps CPU, en Ko pour les autres limites). Si on ajoute l'option `-h`, on modifie la limite stricte :

```
% limit cputime 240
% limit coredumpsize 100
% limit
```

```
cputime         4:00
filesize        unlimited
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    100 kbytes
memoryuse       unlimited
descriptors     1024
memorylocked    unlimited
maxproc         256
openfiles       1024
% limit -h coredumpsize 1000
% limit -h coredumpsize 2000
limit: coredumpsize: Can't set hard limit
% limit -h
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       unlimited
coredumpsize    1000 kbytes
memoryuse       unlimited
descriptors     1024
memorylocked    unlimited
maxproc         256
openfiles       1024
% limit coredumpsize 4000
limit: coredumpsize: Can't set limit
%
```

Cet exemple nous montre bien qu'on ne peut que réduire les limites « hard » (la tentative de ramener `coredumpsize` stricte à 2000 échoue). De même, une limite souple ne peut pas être programmée au-dessus de la limite stricte (c'est pareil pour `coredumpsize` à 4000).

Sous *bash*, la commande est `ulimit`. Elle peut être suivie de `-S` (par défaut) ou de `-H`, pour indiquer une limite souple ou stricte, puis d'une lettre s'appliquant au type de ressource recherchée, et éventuellement de la nouvelle valeur. Les lettres correspondant aux limites sont :

Option	Signification
a	Toutes les limites (affichage seulement)
c	Taille d'un fichier `core`
d	Taille du segment de données d'un processus
f	Taille maximale d'un fichier
m	Taille maximale des données se trouvant simultanément en mémoire
s	Taille de la pile
t	Temps processeur maximal
n	Nombre de fichiers ouverts simultanément
u	Nombre maximal de processus par utilisateur
v	Quantité de mémoire virtuelle disponible

Voici les mêmes manipulations que sous *tcsh*, effectuées cette fois-ci sous *bash*.

```
$ ulimit -a
core file size (blocks)  1000000
data seg size (kbytes)   unlimited
file size (blocks)       unlimited
max memory size (kbytes) unlimited
stack size (kbytes)      8192
cpu time (seconds)       unlimited
max user processes       256
pipe size (512 bytes)    8
open files               1024
virtual memory (kbytes)  2105343
$ ulimit -Ha
core file size (blocks)  unlimited
data seg size (kbytes)   unlimited
file size (blocks)       unlimited
max memory size (kbytes) unlimited
stack size (kbytes)      unlimited
cpu time (seconds)       unlimited
max user processes       256
pipe size (512 bytes)    8
open files               1024
virtual memory (kbytes)  4194302
$
```

Manifestement, sur notre machine, les fichiers d'initialisation de *bash* (`/etc/profile`) et de *tcsh* (`/etc/csh.cshrc`) sont configurés avec les mêmes limitations pour les fichiers `core`.

```
$ ulimit -t 240
$ ulimit -c 100
$ ulimit -tc
core file size (blocks)  100
cpu time (seconds)       240
$ ulimit -Hc 1000
$ ulimit -Hc 2000
ulimit: cannot raise limit: Opération non permise
$ ulimit -Hc
1000
$ ulimit -c 4000
ulimit: cannot raise limit: Opération non permise
$
```

On retrouve évidemment les mêmes restrictions quand on tente de relever une limite forte ou d'augmenter une limite souple au-dessus de la valeur stricte correspondante.

Nous allons à présent étudier comment consulter et modifier une ressource à partir d'un programme C. La fonction `getrlimit()` dont nous avons fourni le prototype plus haut nous remplit une structure `rlimit`, contenant les membres suivants :

Type	Nom	Signification
`rlim_t`	`rlim_cur`	Limite souple (valeur actuelle)
`rlim_t`	`rlim_max`	Limite stricte

Le type de donnée `rlim_t` est définie sous Linux, avec la GlibC , sur architecture PC comme un `long int`. Sur certaines machines, il peut toutefois s'agir d'un `long long int`. On peut donc utiliser un format d'affichage `long long` pour être tranquille. Voici un programme qui affiche les limites strictes, suivies des limites souples entre parenthèses.

exemple_getrlimit.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/time.h>
    #include <sys/resource.h>

    void affichage_limite (char * libelle, int numero);

    int
main (void)
{
    affichage_limite("temps CPU en secondes      ", RLIMIT_CPU);
    affichage_limite("taille maxi d'un fichier   ", RLIMIT_FSIZE);
    affichage_limite("taille maxi zone de données", RLIMIT_DATA);
    affichage_limite("taille maxi de la pile     ", RLIMIT_STACK);
    affichage_limite("taille maxi fichier core   ", RLIMIT_CORE);
    affichage_limite("taille maxi résidente      ", RLIMIT_RSS);
    affichage_limite("nombre maxi de processus   ", RLIMIT_NPROC);
    affichage_limite("nombre de fichiers ouverts ", RLIMIT_NOFILE);
    affichage_limite("taille mémoire verrouillée ", RLIMIT_NOFILE);
    return EXIT_SUCCESS;
}

    void
affichage_limite (char * libelle, int numero)
{
    struct rlimit limite;
    if (getrlimit(numero, & limite) != 0) {
        fprintf(stdout, "Impossible d'accéder à la limite de %s\n",
                libelle);
        return;
    }
    fprintf(stdout, "Limite de %s : ", libelle);
    if (limite.rlim_max == RLIM_INFINITY)
        fprintf(stdout, "illimitée ");
    else
        fprintf(stdout, "%lld ", (long long int) (limite . rlim_max));
    if (limite.rlim_cur == RLIM_INFINITY)
        fprintf(stdout, "(illimitée)\n");
    else
        fprintf(stdout, "(%lld)\n", (long long int) (limite . rlim_cur));
}
```

Voici un exemple d'exécution, qui nous fournit les mêmes résultats que les précédents exemples, directement depuis le shell.

```
$ ./exemple_getrlimit
Limite de temps CPU en secondes       : illimitée (illimitée)
Limite de taille maxi d'un fichier    : illimitée (illimitée)
Limite de taille maxi zone de données : illimitée (illimitée)
Limite de taille maxi de la pile      : illimitée (8388608)
Limite de taille maxi fichier core    : illimitée (1024000000)
Limite de taille maxi résidente       : illimitée (illimitée)
Limite de nombre maxi de processus    : 256 (256)
Limite de nombre de fichiers ouverts  : 1024 (1024)
Limite de taille mémoire verrouillée  : 1024 (1024)
$
```

La fonction **`setrlimit()`** permet de fixer une limite, avec les restrictions que nous avons vues avec les shells. Le prototype de la fonction est :

```
int setrlimit (int ressource, struct rlimit * limite);
```

Attention

Lorsqu'on désire modifier par exemple la limite souple, il est nécessaire de lire auparavant l'ensemble de la structure `rlimit` correspondante afin d'avoir la bonne valeur pour la limite stricte. En effet, les deux champs doivent être correctement renseignés.

On remarquera que le fait de diminuer une limite stricte ne réduit pas nécessairement la limite souple correspondante. On peut temporairement se retrouver avec une limite souple supérieure à la limite stricte. Par contre, à la tentative suivante de modification de la limite souple, elle sera soumise à la nouvelle restriction.

Il faut également savoir que même *root* ne peut pas augmenter le nombre de fichiers ouverts simultanément (`RLIMIT_NOFILE`) au-dessus de la limite imposée par le noyau (`NR_OPEN` défini dans `<linux/limits.h>`).

L'exemple que nous allons développer sert à éviter la création de fichier `core` au cas où le programme plante. Pour simuler un bogue, nous allons nous envoyer le signal `SIGSEGV` (violation mémoire), qui arrête le programme avec, en principe, la création d'une image mémoire sur le disque. Nous allons, pour affiner encore cette gestion de la phase de débogage, encadrer la suppression des fichiers `core` par des directives `#ifdef`-`#endif` concernant la constante symbolique `NDEBUG`. Le fait de définir cette constante permet, rappelons-nous, d'éliminer du programme toutes les macros `assert()` qui servent à surveiller les conditions de fonctionnement du programme. Ainsi, si on ne définit pas la constante, les assertions seront incorporées, et en cas d'arrêt anormal du programme, l'image mémoire `core` servira à un débogage post-mortem du processus. De même, lorsqu'on définira la constante `NDEBUG`, on basculera en code de production, supprimant les assertions, et ramenant à zéro la taille limite des fichiers `core`.

exemple_setrlimit.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <signal.h>
    #include <sys/resource.h>

    int
main (void)
{
#ifdef NDEBUG
    struct rlimit limite;
    if (getrlimit(RLIMIT_CORE, & limite) != 0) {
        fprintf(stderr, "Impossible d'accéder à RLIMIT_CORE\n");
        return EXIT_FAILURE;
    }
    limite.rlim_cur = 0;
    if (setrlimit (RLIMIT_CORE, & limite) != 0) {
        fprintf(stderr, "Impossible d'écrire RLIMIT_CORE\n");
        return EXIT_FAILURE;
    }
    fprintf(stdout, "Code définitif, \"core\" évité \n");
#else
    fprintf(stdout, "Code de développement, \"core\" créé si besoin \n");
#endif

    /*
     * Et maintenant... on se plante !
     */
    raise(SIGSEGV);
    return EXIT_SUCCESS;
}
```

Voici des exemples d'exécution du programme en fonction des différentes directives de compilation :

```
$ cc -Wall exemple_setrlimit.c -o exemple_setrlimit
$ ./exemple_setrlimit
Code de développement, "core" créé si besoin
Segmentation fault (core dumped)
$ rm core
$ cc -Wall -DNDEBUG exemple_setrlimit.c -o exemple_setrlimit
$ ./exemple_setrlimit
Code définitif, "core" évité
Segmentation fault
$ ls core
ls: core: Aucun fichier ou répertoire de ce type
$
```

Conclusion

Lorsqu'un processus doit se mettre en attente pendant une période déterminée, il existe plusieurs méthodes de sommeil avec des durées plus ou moins précises, que nous avons étudiées en détail dans ce chapitre. Nous reviendrons plus longuement sur les notions concernant la date et l'heure dans le chapitre 25.

Nous avons vu que les sommeils et les temporisations sont limités par la précision de l'ordonnanceur, soit 1 milliseconde pour les noyaux 2.6. Lorsqu'il est nécessaire d'avoir une meilleure précision, il faudra se tourner vers des versions de Linux modifiées pour le temps-réel strict (*hard realtime*) comme RTAI.

Nous avons également observé les fonctions permettant d'obtenir des informations sur l'utilisation des ressources système par un processus et de limiter cet accès aux ressources afin de ne pas léser les autres utilisateurs.

10

Entrées-sorties simplifiées

Flux standard d'un processus

Les entrées-sorties sous Linux sont uniformisées par l'intermédiaire de fichiers. Nous verrons, dans la partie consacrée à la gestion des fichiers, qu'on peut y accéder grâce à des primitives de bas niveau (des appels-système) gérant des descripteurs ou par des fonctions de haut niveau (de la bibliothèque C) manipulant des flux.

Les flux sont une abstraction ajoutant automatiquement aux descripteurs de fichiers des tampons d'entrée-sortie, des verrous, ainsi que des rapports d'état et d'erreur plus fins. Les flux sont du type opaque `FILE`, défini dans `<stdio.h>` (ou plutôt dans `<libio.h>`, inclus par ce dernier). On ne doit pas tenter d'accéder aux membres internes de la structure `FILE`, pas plus qu'on ne doit utiliser d'objets de type `FILE`, mais uniquement des pointeurs sur ces objets. Les allocations et libérations de mémoire nécessaires sont entièrement gérées par les fonctions de la bibliothèque C.

Lorsqu'on désire accéder à un fichier par l'intermédiaire d'un flux, on invoque la fonction `fopen()`, que nous décrirons plus en détail dans le chapitre 18. Cette fonction prend en argument le nom du fichier désiré, ainsi qu'une chaîne de caractères indiquant le mode d'accès voulu, et renvoie un pointeur sur un flux de type `FILE *`. On l'utilise ainsi

```
FILE * fp;

fp = fopen ("mon_fichier.txt", "r");
```

pour ouvrir le fichier en lecture seule (mode `r` – *read*). Le pointeur de flux renvoyé peut alors être utilisé pour lire des données. Si on ouvrait notre fichier en mode écriture `w`, *write*, on pourrait alors y écrire des informations.

Nous détaillerons toutes ces notions plus loin, mais ce qui nous intéresse pour l'instant c'est que tout programme s'exécutant sous Linux dispose de trois flux ouverts automatiquement lors de son démarrage.

Ces trois flux sont déclarés dans `<stdio.h>` :

- `stdin` : flux d'entrée standard. Ce flux est ouvert en lecture seule ; il s'agit par défaut du clavier. Le processus peut y recevoir ses données.
- `stdout` : flux de sortie standard. Ouvert en écriture seule, le processus y affiche ses résultats normaux. Par défaut, il s'agit de l'écran de l'utilisateur.
- `stderr` : flux d'erreur standard. Ce flux, ouvert en écriture seule, sert à afficher des informations concernant le comportement du processus ou ses éventuels problèmes. Par défaut, ces informations sont également affichées sur l'écran de l'utilisateur.

Nous arrivons ici à l'un des principes de la conception même des systèmes Unix. Dans cet environnement, une grande partie des outils et des commandes de base sont vus comme des filtres. Ils reçoivent des données en entrée, les transforment, et fournissent leurs résultats en sortie. En cas de problème, l'utilisateur est averti par un message qui s'affiche sur un flux de sortie distinct.

Il est possible, au niveau du shell, de rediriger les flux d'entrée ou de sortie d'un processus à volonté. On peut rediriger par exemple la sortie d'un programme vers un fichier en utilisant l'opérateur `>` :

```
$ mon_programme > sortie.txt
```

À ce moment, toutes les données écrites sur `stdout` seront envoyées dans le fichier concerné. Les informations relatives à `stderr` resteront sur l'écran.

On peut également rediriger l'entrée standard pour lire les données depuis un fichier avec l'opérateur `<` ;

```
$ mon_programme < entree.txt
```

On peut orienter la sortie standard d'un processus directement vers l'entrée standard d'un autre en utilisant l'opérateur `|` :

```
$ programme_1 | programme_2
```

Cette opération est souvent effectuée pour renvoyer les données vers un utilitaire de pagination comme `more` ou `less`. Dans ce dernier cas également, les informations de diagnostic provenant de `programme_1` et envoyées sur `stderr` sont affichées à l'écran, et ne sont pas mêlées aux données de `stdout`, qui sont dirigées vers `programme_2`.

Pareillement, il est possible de procéder à d'autres interventions, comme ajouter la sortie standard en fin de fichier sans écrasement, regrouper la sortie d'erreur et la sortie standard, lire des données directement depuis la ligne de commande ou le script shell utilisés. Ces opérateurs peuvent varier suivant le shell employé. On se reportera, pour plus de détails, à un ouvrage traitant de la programmation sous shell ou à la page de manuel de l'interpréteur concerné.

En fait, les notions d'entrée standard, de sortie standard et de sortie d'erreur sont des concepts de l'univers Unix qui n'ont pas de réelle signification au niveau du noyau. Il s'agit simplement d'une convention instituée par les shells historiques (et qui risque fort de tomber peu à peu en désuétude avec l'avènement des environnements uniquement graphiques).

Dans ce chapitre, nous allons nous intéresser à la présentation ou à la lecture de données sous forme de textes, lisibles par un être humain. Toutefois, les flux standard d'un processus peuvent également être utilisés pour transporter des données binaires. Cela permet de construire une série de petits outils indépendants qu'on regroupe ensuite. Lorsque nous aborderons la programmation réseau, nous étudierons un programme capable de recevoir des données sur un port réseau UDP/IP et de les écrire sur sa sortie standard. De même, un autre programme lira son entrée standard et enverra les informations vers le port UDP/IP d'une autre application. Ces programmes s'appelleront respectivement `udp_2_stdout` et `stdin_2_udp` (le `2` doit se lire *two*, ou plutôt *to*, c'est-à-dire « vers » ; c'est une tradition de nommer ainsi les programmes de filtrage servant à changer le format de leurs données).

Imaginons qu'on ait un autre programme nommé `convertisseur`, qui modifie les informations reçues sur son entrée standard pour les renvoyer en sortie. Avec ces trois outils, nous pouvons obtenir toutes les possibilités suivantes :

- Enregistrement de données provenant d'une application serveuse :

```
$ udp_2_stdout > fichier_1
```

- Relecture des données et émission vers l'application cliente :

```
$ stdin_2_udp < fichier_1
```

- Passerelle entre deux réseaux, par exemple :

```
$ udp_2_stdout | stdin_2_udp
```

- Conversion des données « à froid » :

```
$ convertisseur < fichier_1 > fichier_2
```

- Conversion « au vol » des données :

```
$ udp_2_stdout | convertisseur | stdin_2_udp
```

- On peut les insérer dans un filtre entre deux programmes :

```
$ programme_1 | programme_2
```

qu'on transforme en :

```
$ programme_1 | stdin_2_udp
```

sur une machine, et :

```
$ udp_2_stdout | programme_2
```

sur une autre machine.

Nous voyons la puissance des redirections des flux standard des processus. Ces exemples ne sont pas artificiels, je les ai personnellement utilisés dans une application industrielle pour convertir au vol des données provenant d'un radar et les rendre compatibles avec une application de visualisation se trouvant sur une autre machine. Le fait de pouvoir enregistrer des données ou intercaler un programme d'affichage hexadécimal des valeurs en modifiant simplement le script shell de lancement aide grandement à la mise au point du système.

Écriture formatée dans un flux

L'une des tâches premières des programmes informatiques est d'afficher des messages lisibles par les utilisateurs sur un périphérique de sortie, généralement l'écran. La preuve en est donnée dans le célèbre *hello.c* [KERNIGHAN 1994], dont l'unique rôle est d'afficher « *hello world !* » et de se terminer normalement[1].

```
    #include <stdio.h>
main()
{
    printf("Hello, world \n");
}
```

Lorsqu'il s'agit d'une chaîne de caractères constante, comme dans ce fameux exemple, le travail est relativement simple – il suffit d'envoyer les caractères l'un après l'autre sur le flux de sortie –, mais les choses se compliquent nettement quand il faut afficher des valeurs numériques. La conversion entre la valeur 2005, contenue dans une variable de type `int`, et la série de caractères '2', '0', '0' et '5' n'est déjà pas une tâche simple. Ce qui se présente comme un exercice classique des premiers cours d'assembleur se corse nettement lorsqu'il faut gérer les valeurs signées, puis différentes bases d'affichage (décimal, hexa, octal). Imaginez alors la complexité du travail qui est nécessaire pour afficher le contenu d'une variable en virgule flottante, avec la multitude de formats possibles et le nombre de chiffres significatifs adéquat.

Heureusement, la bibliothèque C standard nous offre les fonctions de la famille `printf()`, qui permettent d'effectuer automatiquement les conversions requises pour afficher les données. Ces routines sont de grands classiques depuis les premières versions des bibliothèques standard du langage C, aussi nous ne détaillerons pas en profondeur chaque possibilité de conversion. On pourra, pour avoir plus de renseignements, se reporter à la page de manuel `printf(3)`.

Il existe quatre variantes sur le thème de `printf()`, chacune d'elles étant disponible en deux versions, suivant la présentation des arguments.

La fonction la plus utile est bien souvent **`fprintf()`**, dont le prototype est déclaré dans `<stdio.h>` ainsi :

```
int fprintf (FILE * flux, const char * format, ...);
```

Les points de suspension indiquent qu'on peut fournir un nombre variable d'arguments à cet emplacement. Le premier argument est le flux dans lequel on veut écrire ; il peut s'agir bien entendu de `stdout` ou `stderr`, mais nous verrons ultérieurement qu'il peut s'agir aussi de n'importe quel fichier préalablement ouvert avec la fonction `fopen()`.

Le second argument est une chaîne de caractères qui sera envoyée sur le flux indiqué, après avoir remplacé certains caractères spéciaux qu'elle contient. Ceux-ci indiquent la conversion à apporter aux arguments situés à la fin de l'appel avant de les afficher. Par exemple, la séquence `%d` dans le format sera remplacée par la représentation décimale de l'argument de type entier situé à la suite du format. On peut bien entendu placer plusieurs arguments à afficher, en indiquant dans la chaîne de format autant de caractères de conversion.

1. L'absence de `return(0)` n'est pas un oubli, mais est due à la volonté de reproduire exactement l'exemple original de Kernighan et de Ritchie.

Les conversions possibles avec la GlibC sont les suivantes :

Conversion	But
`%d`	Afficher un nombre entier sous forme décimale signée.
`%i`	Synonyme de `%d`.
`%u`	Afficher un nombre entier sous forme décimale non signée.
`%o`	Afficher un nombre entier sous forme octale non signée.
`%x`	Afficher un entier non signé sous forme hexa avec des minuscules.
`%X`	Afficher un entier non signé sous forme hexa avec des majuscules.
`%f`	Afficher un nombre réel en notation classique (3.14159).
`%e`	Afficher un réel en notation ingénieur (1.602e-19).
`%E`	Afficher un réel en notation ingénieur avec E majuscule.
`%g`	Afficher un réel le plus lisiblement possible entre `%f` et `%e` suivant sa valeur.
`%G`	Comme `%g`, mais en choisissant entre `%f` et `%E`.
`%a`	Afficher un réel avec la mantisse en hexa et l'exposant de 2 en décimal.
`%A`	Comme `%a`, mais le « P » indiquant l'exposant de 2 est en majuscule.
`%c`	Afficher un simple caractère.
`%C`	Afficher un caractère large (voir chapitre 23).
`%s`	Afficher une chaîne de caractères.
`%S`	Afficher une chaîne de caractères larges.
`%p`	Afficher la valeur d'un pointeur.
`%n`	Mémoriser le nombre de caractères déjà écrits (voir plus bas).
`%m`	Afficher la chaîne de caractères décrivant le contenu de `errno`.
`%%`	Afficher le caractère de pourcentage.

Notons tout de suite que `%m` est une extension Gnu, qui correspond à afficher la chaîne de caractères `strerror(errno)`. Signalons également que la conversion `%n` est très particulière puisqu'elle n'écrit rien en sortie, mais stocke dans l'argument correspondant, qui doit être un pointeur de type `int *`, le nombre de caractères qui a déjà été envoyé dans le flux de sortie. Cette fonctionnalité n'est pas couramment employée ; on peut imaginer l'utiliser avec `sprintf()`, que nous verrons ci-dessous, pour mémoriser l'emplacement des champs de données successifs si on désire y accéder à nouveau par la suite.

La conversion `%n` peut introduire des failles de sécurité par écrasement de mémoire. Si l'utilisateur peut fixer lui-même le format de sortie de `printf()`, il dispose d'un moyen d'écrire dans la mémoire du processus et de lui faire ainsi exécuter n'importe quel code. Ceci est très dangereux pour un programme Set-UID ou un démon réseau. L'affichage avec `printf()` d'une chaîne de caractères fournie par l'utilisateur se fera avec `printf("%s", chaine)` et jamais `printf(chaine)`.

Les autres conversions sont très classiques en langage C et ne nécessitent pas plus de détails ici. Voyons un exemple d'utilisation des diverses conversions.

exemple_fprintf_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <limits.h>

    int
main (void)
{
    int          d = INT_MAX;
    unsigned int u = UINT_MAX;
    unsigned int o = INT_MAX;
    unsigned int x = UINT_MAX;
    unsigned int X = UINT_MAX;
    double       f = 1.04;
    double       e = 1500;
    double       E = 101325;
    double       g = 1500;
    double       G = 0.00000101325;
    double       a = 1.0/65536.0;
    double       A = 0.125;
    char         c = 'a';
    char       * s = "chaîne";
    void       * p = (void *) main;

    fprintf(stdout, " d=%d \n u=%u \n o=%o \n x=%x \n X=%X \n"
                    " f=%f \n e=%e \n E=%E \n g=%g \n G=%G \n"
                    " a=%a \n A=%A \n c=%c \n s=%s \n p=%p \n",
                    d, u, o, x, X, f, e, E, g, G, a, A, c, s, p);

    return EXIT_SUCCESS;
}
```

Bien sûr, les valeurs `INT_MAX` et `UINT_MAX` définies dans `<limits.h>` peuvent varier avec l'architecture de la machine.

```
$ ./exemple_fprintf_1
 d=2147483647
 u=4294967295
 o=17777777777
 x=ffffffff
 X=FFFFFFFF
 f=1.040000
 e=1.500000e+03
 E=1.013250E+05
 g=1500
 G=1.01325E-06
 a=0x1p-16
 A=0X1P-3
 c=a
 s=chaîne
 p=0x80483f0
$
```

On peut incorporer, comme dans n'importe quelle chaîne de caractères en langage C, des caractères spéciaux comme `\n`, `\r`, `\t`…, qui seront interprétés par le terminal au moment de l'affichage.

Entre le symbole `%` et le caractère indiquant la conversion à effectuer, on peut insérer plusieurs indications permettant de modifier la conversion ou de préciser le formatage, en termes de largeur minimale ou maximale d'affichage.

Le premier indicateur qu'on peut ajouter concerne le formatage. Il sert principalement à spécifier sur quel côté le champ doit être aligné. À la suite de cet indicateur peut se trouver un nombre signalant la largeur minimale du champ. On justifie ainsi des valeurs en colonne. On peut encore inclure un point, suivi d'une deuxième valeur marquant la précision d'affichage de la valeur numérique. Un dernier modificateur peut être introduit afin de préciser comment la conversion de type doit être effectuée à partir du type effectif de la variable transmise en argument.

Pour les conversions entières (`%d`, `%i`, `%u`, `%o`, `%x`, `%X`) ou réelles (`%f`, `%e`, `%E`, `%g`, `%G`), on peut utiliser – en premier caractère – les indicateurs de formatage suivants :

Caractère	Formatage
+	Toujours afficher le signe dans les conversions signées.
-	Aligner les chiffres à gauche et non à droite.
espace	Laisser un espace avant les chiffres positifs d'une conversion signée.
0 (zéro)	Compléter le chiffre par des zéros au début plutôt que par des espaces à la fin.
#	Préfixer par `0x` ou `0X` les conversions hexadécimales, et par `0` les conversions octales. Le résultat peut ainsi être relu automatiquement.

Avec les conversions affichant un caractère (`%c`), une chaîne (`%s`) ou un pointeur (`%p`), seul l'indicateur « - » peut être utilisé, afin d'indiquer une justification à gauche du champ.

À la suite de ce modificateur, on indique éventuellement la largeur minimale du champ. Si la valeur à afficher est plus longue, elle débordera. Par contre, si elle est plus courte, elle sera alignée à droite ou à gauche, et complétée par des espaces ou par des zéros suivant le formatage vu précédemment.

Après la largeur minimale du champ, on peut placer un point suivi de la précision de la valeur numérique. La précision correspond au nombre minimal de chiffres affichés dans le cas d'une conversion entière et au nombre de décimales lors des conversions de nombres réels. Voici quelques exemples de formatage en colonne.

exemple_fprintf_2.c:

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int d;
```

```
  fprintf(stdout, "| %%6d  | %%+6d | %%-6d | %%-+6d| %% 6d | %%06d |\n");
  fprintf(stdout, "+------+------+------+------+------+------+\n");
  d = 0;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  d = 1;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  d = -2;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  d = 100;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  d = 1000;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  d = 10000;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  d = 100000;
  fprintf(stdout, "|%6d|%+6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
  return EXIT_SUCCESS;
}
```

```
$ ./exemple_fprintf_2
| %6d  | %+6d | %-6d | %-+6d| % 6d | %06d |
+------+------+------+------+------+------+
|     0|    +0|0     |+0    |     0|000000|
|     1|    +1|1     |+1    |     1|000001|
|    -2|    -2|-2    |-2    |    -2|-00002|
|   100|  +100|100   |+100  |   100|000100|
|  1000| +1000|1000  |+1000 |  1000|001000|
| 10000|+10000|10000 |+10000| 10000|010000|
|100000|+100000|100000|+100000| 100000|100000|
$
```

Nous voyons que l'indication de largeur du champ correspond bien à une largeur minimale, un débordement pouvant se produire, comme c'est le cas sur la dernière ligne si le signe est affiché. L'exemple suivant montre l'effet de l'indicateur de précision sur des conversions entières et réelles.

exemple_fprintf_3.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int    d;
    double f;

    fprintf(stdout, "|  %%8.0d |  %%8.2d |  %%8.0f "
                    "|  %%8.2f |  %%8.2e |  %%8.2g |\n");
    fprintf(stdout, "+--------+--------+--------"
                    "+--------+--------+--------+\n");
    d = 0;
```

```
    f = 0.0;
    fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
                    d, d, f, f, f, f);
    d = 1;
    f = 1.0;
    fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
                    d, d, f, f, f, f);
    d = -2;
    f = -2.0;
    fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
                    d, d, f, f, f, f);
    d = 10;
    f = 10.1;
    fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
                    d, d, f, f, f, f);
    d = 100;
    f = 100.01;
    fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
                    d, d, f, f, f, f);
    return EXIT_SUCCESS;
}
```

```
$ ./exemple_fprintf_3
|  %8.0d |  %8.2d |  %8.0f |  %8.2f |  %8.2e |  %8.2g |
+--------+--------+--------+--------+--------+--------+
|        |      00|       0|    0.00|0.00e+00|       0|
|       1|      01|       1|    1.00|1.00e+00|       1|
|      -2|     -02|      -2|   -2.00|-2.00e+00|      -2|
|      10|      10|      10|   10.10|1.01e+01|      10|
|     100|     100|     100|  100.01|1.00e+02|   1e+02|
$
```

Notons là encore que la largeur indiquée peut être dépassée au besoin (comme avec –2 en notation exponentielle). La précision correspond bien au nombre minimal de chiffres affichés pour les entiers et au nombre de décimales pour les réels.

Enfin, le dernier indicateur est un modificateur qui précise le type réel de l'argument transmis, avant sa conversion. Avec les conversions entières, les modificateurs suivants sont autorisés :

Modificateur	Effet
`h`	L'argument est un `short int` ou un `unsigned short int`.
`hh`	L'argument est un `char` ou un `unsigned char`.
`l`	L'argument est un `long int` ou un `unsigned long int`.
`ll`, `L`, ou `q`	L'argument est un `long long int`, parfois nommé « `quad` » sur d'autres systèmes.
`t`	L'argument est de type `ptrdiff_t`.
`z`	L'argument est de type `size_t` ou `ssize_t`.

Le choix entre le type signé ou non dépend du type de conversion qui suit le modificateur (`%d` ou `%u`, par exemple).

`ptrdiff_t` sert lorsqu'on effectue manuellement des opérations arithmétiques sur les pointeurs. Le type `size_t` ou sa version signée `ssize_t` servent à mesurer la taille des données.

Avec les conversions réelles, tout type de donnée est promue au rang de `double` avant d'être affichée. On peut éventuellement utiliser le modificateur `L`, qui indique que l'argument est de type `long double`. Il n'y a pas d'autre modificateur pour les conversions réelles.

Nous allons à présent observer quelques particularités moins connues de `fprintf()` : la largeur de champ variable et la permutation des arguments. Si on remplace la largeur minimale du champ ou la précision numérique par un astérisque, la valeur sera lue dans l'argument suivant de `fprintf()`. Cela permet de fixer la largeur d'un champ de manière dynamique. En voici une démonstration.

exemple_fprintf_4.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int largeur;
    int nb_chiffres;

    for (largeur = 1; largeur < 10; largeur ++)
        fprintf(stdout, "|%*d|\n", largeur, largeur);
    for (nb_chiffres = 0; nb_chiffres < 10; nb_chiffres ++)
        fprintf(stdout, "|%.*d|\n", nb_chiffres, nb_chiffres);
    return EXIT_SUCCESS;
}
```

```
$ ./exemple_fprintf_4
|1|
| 2|
|  3|
|   4|
|    5|
|     6|
|      7|
|       8|
|        9|
||
|1|
|02|
|003|
|0004|
|00005|
|000006|
|0000007|
|00000008|
|000000009|
$
```

L'intérêt de cette caractéristique est principalement de pouvoir fixer la largeur des colonnes d'un tableau pendant l'exécution du programme (éventuellement après avoir vérifié que la plus grande valeur y tient).

La permutation de paramètre est une deuxième particularité peu connue de `fprintf()`. On peut indiquer en tout début de conversion, juste après le symbole `%`, le numéro du paramètre qu'on désire convertir, suivi du signe `$`. Ce numéro doit être supérieur ou égal à 1, et inférieur ou égal au rang du dernier argument transmis. Si on utilise cette possibilité, il faut nécessairement le faire pour toutes les conversions lors de l'appel de `fprintf()`, sinon le comportement est indéfini. L'utilité de cette fonctionnalité est de permettre de préciser l'ordre des arguments au sein même de la chaîne de formatage. Une application évidente est d'ordonner correctement les jours, mois et année de la date en fonction des désirs de l'utilisateur, uniquement en sélectionnant la bonne chaîne de formatage.

exemple_fprintf_5.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>

    int
main (void)
{
    int i;

    char * format[2] =
    { "La date est %3$02d/%2$02d/%1$02d\n",
      "Today is %1$02d %2$02d %3$02d\n" };

    time_t  timer;
    struct tm * date;

    time(& timer);
    date = localtime(& timer);

    for (i = 0; i < 2; i ++)
       fprintf(stdout, format[i],
                       date->tm_year % 100,
                       date->tm_mon + 1,
                       date->tm_mday);

    return EXIT_SUCCESS;
}
```

```
$ ./exemple_fprintf_5
La date est 31/12/04
Today is 04 12 31
$
```

On voit la puissance de cette fonctionnalité, qui permet de profiter de la phase de traduction des messages pour réordonner correctement les champs suivant la localisation.

Autres fonctions d'écriture formatée

Toutes les fonctions de la famille `printf()` renvoient le nombre de caractères écrits en sortie, ou une valeur négative en cas d'erreur. Cette valeur est rarement utilisée, aussi certains programmeurs préfixent tous leurs appels à ces fonctions d'un `(void)` destiné à indiquer aux outils de vérification de code, comme *lint*, que la valeur de retour est volontairement ignorée. Sous Linux, tout cela n'est pas nécessaire car *lclint*, l'outil standard de vérification de code, reconnaît les fonctions de la famille `printf()` et sait que leurs valeurs de retour peuvent être ignorées.

La fonction **`printf()`**, dont le prototype est

```
int printf (const char * format, ...);
```

est exactement équivalente à `fprintf(stdout, format, ...)`. Personnellement, je préfère utiliser systématiquement `fprintf()` et indiquer explicitement, à chaque écriture, dans quel flux (`stdout` ou `stderr`) les données doivent être dirigées. C'est une simple question d'habitude.

La fonction **`sprintf()`** est déclarée ainsi :

```
int sprintf (char * buffer, const char * format, ...);
```

Elle permet d'écrire les données formatées dans la chaîne fournie en premier argument, en ajoutant un caractère nul `\0` à la fin. Ce caractère nul n'est pas compté dans la valeur renvoyée par `sprintf()`.

Avec le développement des applications graphiques dans des environnements fenêtrés, l'utilité de `fprintf()` sur le flux de sortie `stdout` est de plus en plus réduite. Les programmes préfèrent envoyer leur sortie sur des composants graphiques (*widgets*) effectuant l'affichage de manière beaucoup plus esthétique. Il est alors courant de mettre les données en forme dans une chaîne de caractères, qu'on transmet ensuite à la bibliothèque graphique.

La chaîne de caractères envoyée en premier argument doit être assez grande pour contenir toutes les données affichées, y compris le caractère nul final. Il est alors nécessaire de dimensionner correctement cette chaîne, ce qui peut se révéler difficile.

> **Attention**
> Le fait de déborder d'une chaîne lors d'une écriture est l'une des pires choses qui puisse arriver à un programme : non seulement son comportement sera erroné, mais en plus les dysfonctionnements se produiront intempestivement, et les symptômes seront variables. Le programme peut essayer d'écrire en dehors de ses limites d'adressage autorisées, ce qui le conduit à se terminer à cause du signal `SIGSEGV`. Il peut aussi corrompre les données se trouvant au-delà de la chaîne et avoir alors un comportement indéfini. Mais, le plus grave, c'est que cette erreur peut être employée volontairement par un pirate pour créer une faille de sécurité dans le système. Nous reparlerons de ce problème dans le paragraphe consacré à la saisie de chaînes de caractères.

Ce problème ne se pose pas avec `printf()` ou `fprintf()`, car l'écriture dans un flux n'est pas limitée (ou du moins les limites sont gérées par le noyau lors de l'écriture effective, et la fonction échoue proprement).

Il existe une fonction **`snprintf()`**, avec le prototype suivant, permettant de régler en partie le problème :

```
int snprintf (char * buffer, size_t taille, const char * format, ...);
```

Cette fonction écrira au maximum `taille` caractères, y compris le nul final. Comme le caractère nul n'est pas compté dans la valeur de retour, cette valeur doit toujours être strictement inférieure à `taille`. Si le retour est supérieur ou égal à `taille`, `snprintf()` nous indique alors que la chaîne est trop petite et que le formatage a été tronqué. Sur d'autres systèmes, `snprintf()` renvoie –1 en cas de dépassement.

Comme il est difficile de dimensionner au départ la chaîne correctement, on peut procéder en plusieurs étapes. Nous allons construire une routine utilitaire, qui va allouer automatiquement une chaîne de caractères de la dimension nécessaire. La libération de cette chaîne après emploi est sous la responsabilité du programme appelant.

Pour construire cette routine, nous appellerons la routine `vsnprintf()`, dont l'emploi est plus simple dans notre cas. Les routines **`vprintf()`**, **`vfprintf()`**, **`vsprintf()`** et **`vsnprintf()`** fonctionnent exactement comme leurs homologues sans « `v` » initial, mais reçoivent les arguments à afficher dans une table de type `va_list`, et pas sous forme de liste variable d'arguments. Le type `va_list` est défini dans `<stdarg.h>`, ainsi que des macros qui permettent de passer d'une liste variable d'arguments à une table qu'on peut parcourir. Les prototypes de ces quatre autres fonctions de la famille `printf()` sont :

```
int vprintf (const char * format, va_list arguments);
int vfprintf (FILE * flux, const char * format, va_list arguments);
int vsprintf (char * buffer, const char * format, va_list arguments);
int vsnprintf (char * buffer, size_t taille,
                  const char * format, va_list arguments);
```

Au sein d'un programme, il est beaucoup plus simple d'invoquer les routines `printf()` que `vprintf()` lorsqu'on connaît le nombre d'arguments à transmettre lors de l'écriture du programme. Ce cas est bien entendu le plus fréquent. Par contre, il peut arriver qu'on ne sache pas à l'avance quels seront les arguments à transmettre, ni même leur nombre. Cette situation se présente par exemple lorsque la mise en forme et les données à afficher sont choisies dynamiquement par l'utilisateur. Un autre exemple est celui d'une routine qui sert de frontal à `printf()`, en offrant une interface assez similaire pour le programmeur qui l'invoque, mais qui effectue des tâches supplémentaires (comme une vérification des données) avant d'appeler effectivement `printf()`. Il faut alors invoquer la version `v` de cette fonction, en lui passant un tableau construit dynamiquement. Nous avons déjà rencontré la même dualité entre tableaux et listes variables d'arguments dans le chapitre 4, avec les fonctions de la famille `exec()`.

Voici un exemple d'implémentation d'une routine semblable à `sprintf()`, mais qui allouera automatiquement l'espace nécessaire pour écrire toutes les données. Elle tente de faire son allocation par étapes successives de 64 caractères (valeur purement arbitraire). Elle renvoie un pointeur `NULL` en cas d'échec d'allocation mémoire.

exemple_vsnprintf.c :

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * alloc_printf (const char * format, ...);
```

```
    int
main (void)
{
    char * buffer;
    char * seizecars = "0123456789ABCDEF";

    buffer = alloc_printf(" %s %s",
                seizecars, seizecars);
    if (buffer != NULL) {
        fprintf(stdout, "Chaîne de %d caractères \n %s \n",
                        strlen(buffer), buffer);
        free(chaine);
    }
    buffer = alloc_printf(" %s  %s  %s  %s",
                seizecars, seizecars, seizecars, seizecars);
    if (buffer != NULL) {
        fprintf(stdout, "Chaîne de %d caractères \n %s \n",
                        strlen(buffer), buffer);
        free(buffer);
   }
    return EXIT_SUCCESS;
}

    char *
alloc_printf (const char * format, ...)
{
    va_list arguments;
    char * retour = NULL;
    int taille = 64;
    int nb_ecrits;

    va_start(arguments, format);
    while (1) {
        retour = realloc(retour, taille);
        if (retour == NULL)
            break;
        nb_ecrits = vsnprintf(retour, taille, format, arguments);
        if ((nb_ecrits >= 0) && (nb_ecrits < taille))
            break;
        taille = taille + 64;
    }
    va_end(arguments);
    return retour;
}
```

Nous appelons deux fois cette routine. La première, avec deux chaînes de 16 caractères, plus deux caractères blancs, ce qui tient nettement dans la tentative d'allocation initiale de 64 octets. Par contre, lors du second appel, on dépasse volontairement ces 64 caractères pour forcer une réallocation automatique.

Après utilisation de la chaîne renvoyée, on prend soin de libérer la mémoire.

```
$ ./exemple_vsnprintf
Chaîne de 34 caractères
 0123456789ABCDEF 0123456789ABCDEF
Chaîne de 71 caractères
 0123456789ABCDEF  0123456789ABCDEF  0123456789ABCDEF  0123456789ABCDEF
$
```

Cette routine peut être très utile pour remplir une chaîne de caractères avant de la transmettre à une boîte de dialogue d'une bibliothèque graphique. On est assuré qu'aucun débordement de chaîne ne risque de se produire.

Écritures simples de caractères ou de chaînes

La fonction la plus simple pour écrire un unique caractère dans un flux est **`fputc()`**, déclarée ainsi dans `<stdio.h>` :

```
int fputc (int c, FILE * flux);
```

Lorsqu'on passe un caractère en argument à `fputc()`, il est tout d'abord converti en `int` avant l'appel, puis à nouveau transformé en `unsigned char` (prenant donc n'importe quelle valeur entre 0 et `UCHAR_MAX`, normalement 255). Il est ensuite envoyé dans le flux indiqué. Si une erreur se produit, `fputc()` renvoie `EOF`, sinon elle renvoie la valeur du caractère émis.

L'exemple suivant va mettre en relief les divers comportements de `fputc()` en fonction de ses arguments.

exemple_fputc.c :

```
    #include <stdio.h>
    #include <limits.h>

    void    test_fputc (int valeur, FILE * fp);

    int
main (void)
{
    test_fputc('A', stdout);
    test_fputc(65, stdout);
    test_fputc(UCHAR_MAX, stdout);
    test_fputc(-1, stdout);
    test_fputc('A', stdin);
    return EXIT_SUCCESS;
}

    void
test_fputc (int valeur, FILE * fp)
{
    int retour;

    retour = fputc(valeur, fp);
    fprintf(stdout, "\n Écrit : %d, ", valeur);
    fprintf(stdout, "retour = %d ", retour);
```

```
    if (retour == EOF)
        fprintf(stdout, "(EOF)");
    fprintf(stdout, "\n");
}
```

Voici le résultat de l'exécution :

```
$ ./exemple_fputc
A
 Écrit : 65, retour = 65
A
 Écrit : 65, retour = 65
ÿ
 Écrit : 255, retour = 255
ÿ
 Écrit : -1, retour = 255

 Écrit : 65, retour = -1 (EOF)
$
```

Lors du premier appel, le caractère `A` est transformé en sa valeur Ascii entière et est affiché normalement. Dans le second cas, la transformation avait été effectuée manuellement auparavant, il n'y a donc pas de différence. Dans le troisième appel, la valeur `UCHAR_MAX` vaut 255, qui se traduit dans la table de caractères ISO 8859-1 par ce curieux `y` avec un tréma[1].

Dans le quatrième exemple, nous voyons que la valeur entière signée –1 est traduite en son équivalent en caractère non signé, c'est-à-dire `UCHAR_MAX`, qui est affiché également. Nous notons ici que la valeur de retour de `fputc()` est 255, c'est-à-dire la valeur en caractère non signé retransformée en `int`. Nous observons alors une chose importante : toutes les valeurs que `fputc()` renvoie, lorsqu'il réussit, sont comprises entre 0 et `UCHAR_MAX`.

Dans le dernier exemple, nous demandons à `fputc()` d'écrire dans le flux `stdin` qui est exclusivement ouvert en lecture. La fonction échoue donc. Mais par contre, elle nous renvoie –1, qui est la valeur attribuée à la constante symbolique `EOF` dans `<stdio.h>`. Comme nous avons observé qu'en cas de réussite la valeur renvoyée est toujours comprise entre 0 et 255, il n'y a pas d'ambiguïté possible. Nous retrouverons ce comportement dans la fonction de lecture d'un caractère, qui renvoie aussi une valeur négative en cas d'échec.

La seconde fonction de sortie de caractère, **`putc()`**, correspond au prototype suivant :

```
int putc (int valeur, FILE * stream);
```

Elle se comporte exactement comme `fputc()`, mais peut être implémentée sous forme de macro. Elle est donc optimisée, mais peut évaluer plusieurs fois ses arguments. On l'utilisera donc de préférence à `fputc()`, mais en faisant attention à ne pas placer en argument des expressions ayant des effets de bord, comme `putc(table[i++])`.

Lorsque la sortie se fait sur le flux `stdout`, on peut utiliser la fonction **`putchar()`**

```
int putchar (int valeur);
```

1. Cette lettre existe bien en français, mais uniquement dans des noms propres, par exemple Pierre Louÿs ou L'Haÿ-les-roses. On trouvera une intéressante réflexion à ce propos dans [André 1996].

qui est équivalente à `putc(valeur, stdout)`, sans avoir besoin d'évaluer le second argument, et qui est donc encore mieux optimisée.

Lorsqu'on désire écrire une chaîne de caractères complète, on utilise la fonction **`fputs()`**, déclarée ainsi :

```
int fputs (const char * s, FILE * fp);
```

Cette fonction envoie dans le flux mentionné la chaîne de caractères transmise, sans le caractère nul `\0` final et sans ajouter non plus de retour à la ligne.

exemple_fputs.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (int argc, char * argv[])
{
    int     i;

    if (argc == 1) {
        fputs("Pas d'argument \n", stdout);
    } else {
        fputs("Arguments : ", stdout);
        for (i = 1; i < argc; i ++)
            fputs(argv[i], stdout);
        fputs("\n", stdout);
    }
    return EXIT_SUCCESS;
}
```

Cet exemple montre que les arguments vont être écrits les uns à la suite des autres, sans séparation :

```
$ ./exemple_fputs
Pas d'argument
$ ./exemple_fputs aze rty uiop
Arguments : azertyuiop
$
```

Le comportement est assez semblable à celui de `fprintf()`, mais une confusion possible vient souvent du fait que l'argument `flux` est le dernier et non plus le premier.

Pour écrire un message sur `stdout`, on peut utiliser la fonction **`puts()`** suivante :

```
int puts (const char * message);
```

Elle écrit le message sur la sortie standard, sans le caractère nul final, mais en ajoutant automatiquement un retour à la ligne `\n`. En remplaçant tous les `fputs(...,stdout)` par `puts(...)` dans l'exemple précédent, on obtient l'exécution suivante :

```
$ ./exemple_puts aze rty uiop
Arguments :
aze
rty
```

```
uiop

$
```

Nous y trouvons évidemment des retours à la ligne supplémentaires car nous avons laissé tous les `\n` déjà présents dans le code précédent.

Saisie de caractères

Lorsqu'on désire lire un caractère depuis un flux, **`fgetc()`** fonctionne exactement à l'inverse de `fputc()`. Cette fonction est déclarée ainsi dans `<stdio.h>` :

```
int fgetc (FILE * flux);
```

Elle lit un unique caractère comme un `unsigned char` et le renvoie une fois qu'il est converti en `int`. La valeur renvoyée est donc comprise entre 0 et `UCHAR_MAX`. En cas d'échec, la valeur renvoyée est `EOF`. Cette constante symbolique est généralement définie comme égale à –1. Quoi qu'il en soit, cette constante n'est jamais située dans l'intervalle 0 à `UCHAR_MAX`. Il est donc important de lire le résultat de `fgetc()` dans une variable de type `int` et de le comparer avec `EOF` avant de le convertir en `char` si la lecture a réussi.

exemple_fgetc.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int i;

    while ((i = fgetc(stdin)) != EOF)
         fprintf(stdout, " %02X\n", i);
    return EXIT_SUCCESS;
}
```

Nous allons lire les données depuis le terminal. En voici un premier exemple :

```
$ ./exemple_fgetc
a
   61
   0A
b
   62
   0A
c
   63
   0A
abc
   61
   62
   63
   0A
$
```

Nous voyons que le programme affiche d'abord la valeur correspondant au caractère `a`, suivie du retour à la ligne `\n` (0x0A). En effet, le terminal sous Linux n'envoie les données qu'après la validation de toute la ligne avec la touche Entrée. Nous en voyons un exemple lorsque la chaîne `abc` est saisie en entier, puis validée avant que les données ne soient envoyées sur le flux `stdin` du programme.

Pour terminer le programme, il faut faire échouer la lecture en lui envoyant le code `EOF`. Pour cela, on utilise la touche Contrôle-D. Ceci est configuré à l'aide de la commande `stty`. On voit, à la fin de la deuxième ligne d'affichage des résultats, que le caractère de contrôle `eof` est attribué à la touche Contrôle-D.

```
$ stty -a
speed 38400 baud; rows 25; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
   [...]
$
```

Pour pouvoir lire directement les données sans attendre le retour chariot, il faut modifier le comportement du terminal. Nous en verrons une description détaillée dans le chapitre sur la gestion des terminaux. On peut quand même agir au niveau du shell pour modifier le mode de lecture du terminal :

```
$ stty -icanon
$ ./exemple_fgetc
a   61
z   7A
e   65
^D   04

$ stty sane
$
```

La commande `stty -icanon` modifie la gestion du terminal. La fonction `fgetc()` permet alors de lire immédiatement les caractères, sans attendre leur validation par la touche Entrée. Par contre, la touche Contrôle-D ne fait plus échouer la lecture, elle renvoie simplement le code normal de la touche. On arrête le programme en utilisant Contrôle-C, qui envoie le signal `SIGINT`. La commande `stty sane` permet de rétablir le terminal dans un état normal.

Nous reviendrons dans le chapitre consacré à la gestion des terminaux sur le moyen de modifier la configuration du terminal directement depuis l'application, et d'utiliser des lectures non bloquantes pour capturer des caractères au vol, afin que l'application puisse continuer à s'exécuter même si l'utilisateur n'a pas appuyé sur des touches.

On peut employer également **`getc()`**, qui est fonctionnellement équivalente à `fgetc()`, mais qui peut être implémentée sous forme de macro, évaluant plusieurs fois son argument `flux`.

Enfin, la routine **`getchar()`** est équivalente à `getc(stdin)`. Nous allons l'employer pour écrire une application qui affiche le contenu de son entrée standard en hexadécimal et sous forme de caractères. Ce genre d'utilitaire est souvent employé pour le débogage, pour analyser le contenu de fichiers de données binaires.

exemple_getchar.c :

```
#include <stdio.h>
#include <stdlib.h>
```

```
    #include <ctype.h>

    int
main (void)
{
    int  lu;
    char caracteres[17];
    int  emplacement = 0;
    int  rang = 0;

    caracteres[16] = '\0';

    while ((lu = getchar()) != EOF) {
        if ((rang = emplacement % 16) == 0)
            fprintf(stdout, "%08X  ", emplacement % 0xFFFFFFFF);
        fprintf(stdout, "%02X", lu);
        if (rang == 7)
            fprintf(stdout, "-");
        else
            fprintf(stdout, " ");
        if (isprint(lu))
            caracteres[rang] = lu;
        else
            caracteres[rang] = ' ';
        if (rang == 15)
            fprintf(stdout, " %s\n", caracteres);
        emplacement ++;
    }
    while (rang < 15) {
        fprintf(stdout, "   ");
        caracteres[rang] = '\0';
        rang ++;
    }
    fprintf(stdout, " %s\n", caracteres);
    return EXIT_SUCCESS;
}
```

Ce genre de programme peut être utilisé tant sur des fichiers binaires que sur des fichiers de texte :

```
$ ./exemple_getchar < exemple_getchar.c
00000000  0A 09 23 69 6E 63 6C 75-64 65 20 3C 73 74 64 69    #include <stdi
00000010  6F 2E 68 3E 0A 09 23 69-6E 63 6C 75 64 65 20 3C  o.h>  #include <
00000020  63 74 79 70 65 2E 68 3E-0A 0A 09 69 6E 74 0A 6D  ctype.h>   int m
00000030  61 69 6E 20 28 76 6F 69-64 29 0A 7B 0A 09 69 6E  ain (void) {  in
00000040  74 09 6C 75 3B 0A 09 63-68 61 72 09 63 61 72 61  t lu;  char cara
  [...]
00000280  20 2B 2B 3B 0A 09 7D 0A-09 66 70 72 69 6E 74 66   ++;  }  fprintf
00000290  20 28 73 74 64 6F 75 74-2C 20 22 20 25 73 5C 6E   (stdout, " %s\n
000002A0  22 2C 20 63 61 72 61 63-74 65 72 65 73 29 3B 0A  ", caracteres);
000002B0  09 72 65 74 75 72 6E 20-45 58 49 54 5F 53 55 43  return EXIT_SUC
000002C0  43 4E 53 53 3B 0A 7D 0A                          CESS; }
```

```
$ ./exemple_getchar < exemple_getchar
00000000  7F 45 4C 46 01 01 01 00-00 00 00 00 00 00 00 00   ELF
00000010  02 00 03 00 01 00 00 00-A0 83 04 08 34 00 00 00               4
00000020  60 24 00 00 00 00 00 00-34 00 20 00 06 00 28 00  `$      4     (
00000030  1E 00 1B 00 06 00 00 00-34 00 00 00 34 80 04 08          4   4
00000040  34 80 04 08 C0 00 00 00-C0 00 00 00 05 00 00 00  4
  [...]
00003020  79 70 65 5F 62 40 40 47-4C 49 42 43 5F 32 2E 30  ype_b@@GLIBC_2.0
00003030  00 5F 49 4F 5F 73 74 64-69 6E 5F 75 73 65 64 00   _IO_stdin_used
00003040  5F 5F 64 61 74 61 5F 73-74 61 72 74 00 5F 5F 67  __data_start __g
00003050  6D 6F 6E 5F 73 74 61 72-74 5F 5F 00              mon_start__
$
```

Nous avons bien entendu éliminé de nombreuses lignes pour présenter les résultats du programme. Nous réutiliserons cet utilitaire à plusieurs reprises dans le reste de cet ouvrage pour analyser les effets d'autres programmes d'exemple.

Réinjection de caractère

Il peut arriver qu'on veuille en quelque sorte annuler la lecture d'un caractère. Imaginons une routine qui doit lire des caractères uniquement numériques et qui s'arrête dès qu'elle rencontre un caractère ne se trouvant pas dans l'intervalle '0' - '9'. La suite du traitement sera prise en charge par une autre routine, qui agira en fonction du nouveau caractère lu. Une des éventualités serait de toujours conserver le caractère lu dans une variable globale, la lecture ayant toujours un caractère d'avance sur le traitement proprement dit. C'est d'ailleurs la méthode généralement employée par les analyseurs lexicaux, qui fonctionnent avec des mots complets (*token*).

Une autre possibilité serait de replacer dans le flux d'entrée le dernier caractère lu, pour que la prochaine lecture le renvoie à nouveau. Comme les flux fonctionnent en utilisant des mémoires tampons, il ne s'agit pas d'une véritable écriture dans le fichier associé, mais simplement d'un ajout en tête de buffer. La routine assurant cette tâche est **`ungetc()`**, déclarée ainsi dans `<stdio.h>` :

```
int ungetc (int caractere_lu, FILE * flux);
```

Cette routine replace le caractère transmis dans le flux. Le premier argument est de type `int`, car on peut également lui transmettre la constante symbolique `EOF`. Cela permet au besoin de transmettre directement à `ungetc()` le résultat de `fgetc()`. Il n'est possible de replacer dans le flux qu'un seul caractère, et il est inutile d'invoquer plusieurs fois de suite `ungetc()`. Le comportement est indéfini pour ce qui est de savoir si le dernier caractère transmis écrasera les précédents. Notons également que le caractère qu'on replace dans le flux n'est pas nécessairement celui qu'on vient de lire.

L'exemple que nous allons construire est un peu artificiel : deux routines sont chargées de lire caractère par caractère l'entrée standard. L'une s'occupe des caractères numériques, l'autre des caractères alphabétiques. La routine `main()` centrale lit un caractère puis, s'il correspond à l'une des deux classes de caractères définies ci-dessus, elle réinjecte le caractère lu dans le flux d'entrée, et invoque la routine spécialisée correspondante. Ces routines sont construites de manière à lire tout ce qui arrive puis, dès qu'un caractère ne leur convient pas, elles le replacent dans le flux, et reviennent à la fonction `main()`.

exemple_ungetc.c :

```
    #include <ctype.h>
    #include <stdio.h>

    void lecture_numerique    (FILE * fp);
    void lecture_alphabetique (FILE * fp);

    int
main (void)
{
    int     c;

    while ((c = getc(stdin)) != EOF) {
        if (isdigit(c)) {
            ungetc(c, stdin);
            lecture_numerique(stdin);
        } else if (isalpha(c)) {
            ungetc(c, stdin);
            lecture_alphabetique(stdin);
        }
    }
    return EXIT_SUCCESS;
}

    void
lecture_numerique (FILE * fp)
{
    int     c;
    fprintf(stdout, "Lecture numérique : ");
    while (1) {
        c = getc(fp);
        if (! isdigit(c))
            break;
        fprintf(stdout, "%c", c);
    }
    ungetc(c, fp);
    fprintf(stdout, "\n");
}

    void
lecture_alphabetique (FILE * fp)
{
    int     c;
    fprintf(stdout, "Lecture alphabétique : ");
    while (1) {
        c = getc(fp);
        if (! isalpha(c))
            break;
        fprintf(stdout, "%c", c);
    }
    ungetc(c, fp);
    fprintf(stdout, "\n");
}
```

Voici un exemple d'exécution :

```
$ ./exemple_ungetc
AZE123 ABCDEF9875XYZ
Lecture alphabétique : AZE
Lecture numérique : 123
Lecture alphabétique : ABCDEF
Lecture numérique : 9875
Lecture alphabétique : XYZ
$
```

Saisie de chaînes de caractères

Pour lire une chaîne de caractères, il existe deux fonctions : `gets()` et `fgets()`. Le prototype de **`gets()`** est le suivant :

```
char * gets (char * buffer);
```

Cette fonction lit l'entrée standard `stdin` et place les caractères dans la chaîne passée en argument. Lorsqu'elle rencontre le caractère `EOF` ou un retour chariot, elle les remplace par le caractère nul de fin de chaîne '`\0`', et renvoie le pointeur sur la chaîne. Si le caractère `EOF` est rencontré avant qu'elle ait pu lire un seul caractère, `gets()` renvoie le pointeur `NULL`.

> **Attention**
> Il ne faut jamais utiliser `gets()` !

En effet, `gets()` ne permet pas de préciser la longueur maximale de la chaîne à saisir. En conséquence, si le nombre de caractères reçus excède la taille de la zone qu'on a allouée, `gets()` continuera joyeusement à écrire en mémoire en provoquant un débordement de buffer.

`gets()` servant généralement à lire une chaîne de caractères tapée au clavier par l'utilisateur, on pourrait croire qu'allouer un buffer suffisamment grand éviterait tout problème. Malheureusement, il suffit de rediriger l'entrée standard du processus en provenance d'un fichier pour que la saisie puisse prendre n'importe quelle longueur.

Dans le meilleur des cas, le programme ira écrire en dehors de son espace d'adressage autorisé par le noyau et sera alors terminé par un signal `SIGSEGV`. Mais un grave problème de sécurité peut aussi survenir si le programme est installé avec un bit Set-UID ou Set-GID. En C, les données automatiques des fonctions (celles qui ne sont pas déclarées statiques) sont allouées dans la pile. Lors de l'entrée dans une fonction, l'adresse de retour et les arguments sont empilés. Ensuite, on réserve dans la pile la place nécessaire aux variables automatiques. Par exemple, lors de l'entrée dans la routine suivante

```
    int
fonction (int x)
{
    int  x1;
    char chaine_1[128];
    ...
}
```

on stocke successivement sur la pile l'argument `x`, l'adresse de retour, puis on réserve 4 octets pour `x1`, et 128 octets pour la chaîne. Sous Linux X86, la pile croît vers le bas, signifiant que l'adresse de `chaine_1[0]` est plus petite que celle de `x1`, qui est elle-même inférieure à l'adresse de retour et à l'adresse de `x`. Voici un exemple pour clarifier la situation :

exemple_pile.c :

```
    #include <stdio.h>

    int fonction (int x);

    int
main (void)
{
    return fonction(1);
}

    int
fonction (int x)
{
    int x1;
    char chaine[128];

    fprintf(stdout, "& x    = %p lg = %d\n", & x, sizeof(x));
    fprintf(stdout, "& x1   = %p lg = %d\n", & x1, sizeof(x1));
    fprintf(stdout, "chaine = %p lg = %d\n", chaine, sizeof(chaine));
    if (x > 0)
        return fonction(x - 1);
    return EXIT_SUCCESS;
}
```

La fonction s'appelle elle-même une fois, pour pouvoir déduire la position de l'adresse de retour par rapport à l'argument. Lors de l'exécution, nous obtenons ceci :

```
$ ./exemple_pile
& x    = 0xbffffcf4 lg = 4
& x1   = 0xbffffce8 lg = 4
chaine = 0xbffffc68 lg = 128
& x    = 0xbffffc64 lg = 4
& x1   = 0xbffffc58 lg = 4
chaine = 0xbffffbd8 lg = 128
$
```

Il est clair que lors de la deuxième invocation de `fonction()`, la pile est structurée ainsi :

Adresse début	Adresse fin	Taille	Contenu
0xBFFFFCF4	0xBFFFFCF7	4	Argument `x` (valant 1)
0xBFFFFCEC	0xBFFFFCF3	8	Adresse de retour
0xBFFFFCE8	0xBFFFFCEB	4	Variable automatique `x1`
0xBFFFFC68	0xBFFFFCE7	128	Variable automatique `chaine[]`
0xBFFFFC64	0xBFFFFC67	4	Second argument `x` (valant 0)

Adresse début	Adresse fin	Taille	Contenu
0xBFFFFC5C	0xBFFFFC63	8	Seconde adresse de retour
0xBFFFFC58	0xBFFFFC5B	4	Seconde variable automatique x1
0xBFFFFBD8	0xBFFFFC57	128	Seconde variable automatique chaine[]

Si, lors d'une lecture avec `gets()`, nous débordons de la chaîne allouée dans la pile, nous allons écraser d'abord `x1` – ce qui n'est pas très grave –, mais également l'adresse de retour. Lorsque la fonction va se terminer, le programme va essayer de revenir à une adresse erronée et va avoir un comportement incohérent, difficile à déboguer.

Si le programme est Set-UID, la situation est encore pire car un pirate peut l'exploiter en faisant volontairement déborder la chaîne (en fournissant des données depuis un fichier binaire). Il s'arrangera pour glisser du code valide dans la pile et fera pointer l'adresse de retour sur ce code. Le programme Set-UID exécutera alors exactement ce que veut son utilisateur mais avec l'identité du propriétaire du fichier exécutable. Obtenir un shell *root* est alors très simple. On comprend mieux à présent l'intérêt de limiter les privilèges d'une application Set-UID en diminuant son ensemble de capacités.

Une grande partie des failles de sécurité découvertes dans les programmes Set-UID sont dues à ce genre de problèmes. Il ne faut donc jamais utiliser `gets()`. D'ailleurs, l'éditeur de lien Gnu « `ld` » signale aussi qu'il ne faut pas utiliser cette fonction.

exemple_gets.c :

```
    #include <stdio.h>

    int
main (void)
{
    char chaine[128];

    return (gets(chaine) != NULL);
}
```

Lors de la compilation, on obtient le message suivant :

```
$ cc -Wall exemple_gets.c -o exemple_gets
/tmp/cc5S26rd.o: In function 'main':
/tmp/cc5S26rd.o(.text+0xe): the 'gets' function is dangerous and should not be
used.
$
```

Au contraire, **`fgets()`** est bien plus robuste puisqu'elle permet de limiter la taille de la saisie. Son prototype est le suivant :

```
char * fgets (char * chaine, int taille, FILE * flux);
```

Cette fonction lit les caractères sur le flux indiqué et les place dans la chaîne transmise en argument. En aucun cas, elle ne dépassera `taille-1` caractères lus. Elle s'arrêtera également si elle rencontre un caractère de retour à la ligne '`\n`' ou une fin de fichier `EOF`. Le caractère '`\n`' éventuel est écrit dans le buffer. Ensuite, `fgets()` termine la chaîne par un caractère nul.

Cette routine renvoie le pointeur sur la chaîne passée en argument lorsqu'elle réussit. Au contraire, si elle a rencontré le caractère `EOF` avant d'avoir pu lire quoi que ce soit, elle renvoie un pointeur `NULL`. Si on désire lire une ligne en entier, quelle que soit sa longueur, il est possible d'écrire une routine qui encadre `fgets()` et qui alloue de la mémoire jusqu'à la fin de la ligne. Il s'agit du même genre de fonctionnalité que celle que nous avions créée pour `printf()`.

exemple_fgets.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    char * alloc_fgets (FILE * fp);

    int
main (void)
{
    char * chaine;

    while (1) {
        chaine = alloc_fgets(stdin);
        if (chaine == NULL)
            /* Pas assez de mémoire */
            break;
        if ((chaine[0] == '\n') || (chaine[0] == '\0'))
            /* Chaîne vide... on quitte */
            break;
        fprintf(stdout, "%d caractères \n", strlen(chaine));
        free(chaine);
    }
    return EXIT_SUCCESS;
}

    char *
alloc_fgets (FILE * fp)
{
    char *  retour = NULL;
    char *  a_ecrire = NULL;
    int     taille = 64;

    retour = malloc(taille);
    a_ecrire = retour;

    while (1) {
        if (fgets(a_ecrire, 64, fp) == NULL)
            break;
        if (strlen(a_ecrire) < 63)
            break;
        /* On se place sur le caractère nul final */
        a_ecrire = a_ecrire + 63;
```

```
        /* Et on agrandit également le buffer de 63 caractères */
        taille += 63;
        retour = realloc(retour, taille);
        if (retour == NULL)
            break;
    }
    return retour;
}
```

Le programme lit des chaînes de caractères et en affiche la longueur jusqu'à ce qu'il reçoive une chaîne vide, puis il se termine.

```
$ ./exemple_fgets
ABCDEFGHIJKLMNOPQRSTUVWXYZ
27 caractères
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345678901234567890
73 caractères
ABC
4 caractères

$
```

On remarquera que dans les longueurs affichées, le caractère de retour à la ligne induit par la touche de validation Entrée est comptabilisé. On voit bien qu'avec la chaîne de 73 caractères, la saisie a effectué l'allocation en deux étapes et a bien renvoyé tous les caractères (26 lettres + 26 lettres + 20 chiffres + retour chariot = 73).

La bibliothèque GlibC offre une routine assez semblable, mais qui a l'inconvénient d'être une extension Gnu, donc de ne pas être disponible sur d'autres systèmes. Il est quand même conseillé de s'en servir, et on pourra toujours la redéfinir en utilisant la même méthode que pour notre exemple précédent si le programme doit être porté sur un système différent. Cette routine est nommée **`getline()`**, et elle est déclarée ainsi dans `<stdio.h>` :

```
ssize_t getline (char ** chaine, size_t * taille, FILE * flux);
```

Son utilisation est légèrement moins intuitive, puisqu'elle prend en argument un pointeur sur un pointeur de chaîne de caractères et un pointeur sur une valeur de longueur. Elle tente tout d'abord d'effectuer la lecture dans la chaîne existante, qui doit avoir (`*taille`) octets au moins. Si cela suffit, elle renvoie le nombre de caractères lus, sans compter le caractère nul final qu'elle ajoute. Sinon, elle réalloue de la mémoire, en modifiant le pointeur `chaine` et la taille, jusqu'à ce que la ligne lue tienne en entier dans la chaîne. On peut également l'invoquer avec un pointeur `*chaine` valant `NULL`, et `*taille` valant zéro, elle assurera l'allocation initiale.

La fin de la ligne est déterminée par `EOF` ou par le retour chariot. Si `EOF` arrive dès le début de la lecture ou si une autre erreur se produit, `getline()` renvoie –1 (ce qui explique le type `ssize_t` de la fonction, c'est-à-dire *signed size_t*).

L'avantage de cette routine c'est qu'elle renvoie le nombre de caractères lus. Dans la routine que nous avions écrite, ce nombre ne pouvait être défini qu'à l'aide de `strlen()`. Malheureusement, si la chaîne lue contient un caractère nul, `strlen()` s'arrêtera à ce niveau. Cela peut parfois poser des problèmes lors de la redirection d'un fichier binaire en entrée.

Voyons un exemple d'utilisation de `getline()`, dans le même genre que le précédent :

exemple_getline.c :

```
    #define _GNU_SOURCE

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    int
main (void)
{
    char *  chaine;
    size_t  taille;
    ssize_t retour;

    while (1) {
        taille = 0;
        chaine = NULL;

        retour = getline(& chaine, & taille, stdin);
        if (retour == -1)
            break;

        fprintf(stdout, "%d caractères lus\n", retour);
        fprintf(stdout, "%d caractères alloués \n", taille);
        free(chaine);
    }
    return EXIT_SUCCESS;
}
```

Lors de l'exécution, on arrête le programme en tapant directement sur Contrôle-D (`EOF`) en début de ligne pour provoquer un échec. Nous affichons également la taille du buffer alloué par la routine, afin de pouvoir le dépasser volontairement lors de la seconde saisie.

```
$ ./exemple_getline
ABCDEFGHIJKLMNOPQRSTUVWXYZ
27 caractères lus
120 caractères alloués
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
➥ ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
157 caractères lus
240 caractères alloués
$
```

Nous avons vu comment lire des caractères ou des chaînes. Nous allons à présent nous intéresser à la manière de recevoir des informations correspondant à d'autres types de données.

Lectures formatées depuis un flux

La saisie formatée de données se fait avec les fonctions de la famille `scanf()`. Comme pour la famille `printf()`, il existe six versions ayant les prototypes suivants :

```
int  scanf   (const char * format, ...);
int  vscanf  (const char * format, va_list arguments);

int  fscanf  (FILE * flux, const char * format, ...);
int  vfscanf (FILE * flux, const char * format, va_list arguments);

int  sscanf  (const char * chaine, const char * format, ...)
int  vsscanf (const char * chaine, const char * format, va_list args);
```

Pareillement, il existe en fait trois types de fonctions, chacune disponible en deux versions, avec un nombre variable d'arguments ou avec une table d'arguments. Ce dernier type nécessite l'inclusion du fichier d'en-tête `<stdarg.h>`.

- **`scanf()`** et **`vscanf()`** lisent les données en provenance de `stdin`.
- **`fscanf()`** et **`vfscanf()`** analysent les informations provenant du flux qu'on transmet en premier argument.
- **`sscanf()`** et **`vsscanf()`** effectuent la lecture formatée depuis la chaîne de caractères transmise en premier argument.

L'argument de format se présente comme une chaîne de caractères semblable à celle qui est employée avec `printf()`, mais avec quelques différences subtiles.

Les arguments fournis ensuite sont des pointeurs sur les variables qu'on désire remplir. Les fonctions renvoient le nombre de variables qu'elles ont réussi à remplir correctement.

Contrairement à `printf()`, qui est assez tolérante avec le formatage demandé puisqu'elle assure de toute manière une conversion de type, il faut indiquer ici, dans la chaîne de format, le bon type de donnée correspondant au pointeur à remplir. Si on demande par exemple à `scanf()` de lire un réel `double` et qu'on lui transmette un pointeur sur un `char`, le compilateur fournira un avertissement, mais rien de plus. Lors de l'exécution du programme, l'écriture débordera de la place mémoire réservée au caractère.

Voici un exemple de programme qui plante à coup sûr. Le caractère `c` étant alloué comme une variable automatique de la fonction `main()`, lorsqu'on l'écrase avec une écriture de type `double`, on détruit la pile, y compris l'adresse de retour de `main()` qui est, ne l'oublions pas, une fonction comme les autres avec la simple particularité d'être automatiquement invoquée par le chargeur de programme.

exemple_scanf_1.c :

```
    #include <stdio.h>

    int
main (void)
{
    char c;

    puts("Je vais me planter dès que vous aurez entré un chiffre \n");

    return (scanf("%lf", & c) == 1);
}
```

Et voici le résultat, dont il n'y a pas lieu d'être fier…

```
$ cc -Wall exemple_scanf_1.c -o exemple_scanf_1
exemple_scanf_1.c: In function `main':
exemple_scanf_1.c:11: warning: double format, different type arg (arg 2)
$ ./exemple_scanf_1
Je vais me planter dès que vous aurez entré un chiffre

12
Segmentation fault (core dumped)
$ rm core
$
```

Voyons donc à présent quels sont les bons indicateurs à fournir dans la chaîne de format, en correspondance avec le type de donnée à utiliser.

Type	Format
`char`	`%c`
`char *`	`%s`
`short int`	`%hd %hi`
`unsigned short int`	`%du %do %dx %dX`
`int`	`%d %i`
`unsigned int`	`%u %o %x %X`
`long int`	`%ld %li`
`unsigned long int`	`%lu %lo %lx %lX`
`long long int`	`%lld %Ld %lli %Li`
`unsigned long long int`	`%llu %Lu %llo %Lo %llx %Lx %llX %LX`
`float`	`%e %f %g`
`double`	`%le %lf %lg`
`long double`	`%lle %Le %llf %Lf %llg %Lg`
`void *`	`%p`

Nous voyons qu'il y a en définitive quelques indicateurs principaux et des modificateurs de type. Les indicateurs généraux sont :

Indicateur	Type
`d`	Valeur entière signée sous forme décimale
`i`	Valeur entière signée exprimée comme les constantes en C (avec un préfixe `0` pour l'octal, `0x` pour l'hexadécimal...)
`u`	Valeur entière non signée sous forme décimale
`o`	Valeur entière non signée en octal
`x` ou `X`	Valeur entière non signée en hexadécimal
`e`, `f` ou `g`	Valeur réelle
`s`	Chaîne de caractères sans espace
`c`	Un ou plusieurs caractères

À ceci s'ajoutent les modificateurs `h` pour `short`, `l` pour `long` (dans le cas des entiers) ou pour `double` (dans le cas des réels), et `ll` ou `L` pour `long long` ou pour `long double`. Il existe également des conversions `%C` et `%S` pour les caractères larges, nous arborderons ce sujet dans le chapitre 23.

Notons qu'on peut insérer entre le caractère `%` et l'indicateur de conversion une valeur numérique représentant la taille maximale à accorder à ce champ. Ce détail est précieux avec la conversion `%s` pour éviter un débordement de chaîne. Il est également possible de faire précéder cette longueur d'un caractère '`a`', qui demandera à `scanf()`d'allouer automatiquement la mémoire nécessaire pour la chaîne de caractères à lire. Cela n'a de sens qu'avec une conversion de type `%s`. Dans ce dernier cas, il faut transmettre un pointeur de type `char **`.

L'indicateur de conversion '`c`' est précédé d'une valeur numérique : il indique le nombre de caractères qu'on désire lire. Par défaut, on lit un seul caractère, mais il est ainsi possible de lire des chaînes de taille quelconque. Contrairement à la conversion `s`, la lecture ne s'arrête pas au premier caractère blanc. On peut ainsi lire des chaînes contenant n'importe quel caractère d'espacement. Par contre, `scanf()` n'ajoute pas de caractère nul à la fin de la chaîne, il faut le placer soi-même.

Lorsqu'un caractère non blanc est présent dans la chaîne de format, il doit être mis en correspondance avec la chaîne reçue depuis le flux de lecture. Cela permet d'analyser facilement des données provenant d'autres programmes, si le format d'affichage est bien connu. En voici un exemple :

exemple_scanf_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int i, j, k;

    if (fscanf(stdin, "i = %d j = %d k = %d", & i, & j, & k) == 3)
        fprintf(stdout, "Ok (%d, %d, %d)\n", i, j, k);
    else
        fprintf(stdout, "Erreur \n");
    return EXIT_SUCCESS;
}
```

Ce programme réussit lorsqu'on lui fournit une ligne construite sur le modèle prévu, mais il échoue sinon :

```
$ ./exemple_scanf_2
i=1 j=2 k=3
Ok (1, 2, 3)
$ ./exemple_scanf_2
i = 4   j =   5  k = 006
Ok (4, 5, 6)
$ ./exemple_scanf_2
45 67 89
Erreur
$
```

Ici, les caractères blancs dans la chaîne de format servent à éliminer tous les caractères blancs éventuels présents dans la ligne lue. Les concepteurs de la bibliothèque `stdio` devaient être d'humeur particulièrement facétieuse le jour où ils ont défini le comportement de `scanf()` vis-à-vis des caractères blancs et de la gestion d'erreur. En effet, lorsque `scanf()` reçoit un caractère qu'elle n'arrive pas à mettre en correspondance avec sa chaîne de format, elle le réinjecte dans le flux de lecture avec la fonction `ungetc()`. Ceci se produit par exemple lorsqu'on attend un caractère particulier et qu'un autre arrive, ou lorsqu'on attend un entier et qu'on reçoit un caractère alphabétique. De nombreux débutants en langage C se sont arrachés les cheveux sur le comportement *a priori* incompréhensible de programmes comme celui-ci.

exemple_scanf_3.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int i;

    fprintf(stdout, "Veuillez entrer un entier : ");
    while (1) {
        if (scanf("%d", & i) == 1)
            break;
        fprintf(stdout, "\nErreur, un entier svp :");
    }
    fprintf(stdout, "\nOk\n");
    return EXIT_SUCCESS;
}
```

La saisie se passe très bien tant que l'utilisateur ne commet pas d'erreur :

```
$ ./exemple_scanf_3
Veuillez entrer un entier : 4767

Ok
$
```

Par contre, si on entre un caractère alphabétique à la place d'un chiffre, `scanf()` le refuse, le réinjecte dans le flux et indique qu'elle n'a pu faire aucune conversion. Toute notre belle gestion d'erreur s'effondre alors, car à la tentative suivante nous allons relire à nouveau le même caractère erroné ! Cela se traduit alors par une avalanche de messages d'erreur que seul un Contrôle-C peut interrompre :

```
$ ./exemple_scanf_3
Veuillez entrer un entier : A
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un entier svp :
```

```
Erreur, un entier svp :
Erreur, un entier svp :
   [...]
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un entier svp :
Erreur, un en             (Contrôle-C)
$
```

Le seul moyen simple de gérer ce genre de problème est de passer par une étape de saisie intermédiaire de ligne, à l'aide de la fonction `fgets()`.

exemple_scanf_4.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char    ligne[128];
    int i;

    fprintf(stdout, "Veuillez entrer un entier : ");
    while (1) {
        if (fgets(ligne, 128, stdin) == NULL) {
            fprintf(stderr, "Fin de fichier inattendue \n");
            return EXIT_FAILURE;
        }
        if (sscanf(ligne, "%d", & i) == 1)
            break;
        fprintf(stdout, "\nErreur, un entier svp : ");
    }
    fprintf(stdout, "Ok\n");
    return EXIT_SUCCESS;
}
```

Cette fois-ci, le comportement est celui qu'on attend :

```
$ ./exemple_scanf_4
Veuillez entrer un entier : 12
Ok
$ ./exemple_scanf_4
Veuillez entrer un entier : A

Erreur, un entier svp : Z

Erreur, un entier svp : E

Erreur, un entier svp : 24
Ok
$
```

L'autre piège classique de `scanf()` c'est qu'un caractère blanc dans la chaîne de format élimine tous les caractères blancs présents dans le flux en lecture. Lorsqu'on parle de caractères blancs, il s'agit de l'espace et de la tabulation bien sûr, mais également des retours à la ligne. En fait, il s'agit des caractères correspondant à la fonction `isspace()` que nous verrons dans le chapitre 23, c'est-à-dire l'espace, les tabulations verticales et horizontales '`\t`' et '`\v`', le saut de ligne '`\n`', le retour chariot '`\r`', et le saut de page '`\f`'.

Cela a des conséquences inattendues sur un programme aussi simple que celui-ci.

exemple_scanf_5.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int i;

    fprintf(stdout, "Entrez un entier : ");
    if (scanf("%d", & i) == 1)
        fprintf(stdout, "Ok i=%d\n", i);
    else
        fprintf(stdout, "Erreur \n");
    return EXIT_SUCCESS;
}
```

Tout se passe correctement avec ce programme :

```
$ ./exemple_scanf_5
Entrez un entier : 12
Ok i=12
$ ./exemple_scanf_5
Entrez un entier : A
Erreur
$
```

Par contre, supposons qu'on introduise un caractère blanc supplémentaire à la fin de la chaîne de format. Par exemple, on pourrait en croyant bien faire y ajouter un retour à la ligne '`\n`' pour marquer la fin de la saisie. La ligne de `scanf()` deviendrait :

```
        if (scanf ("%d\n", & i) == 1)
```

Mais le comportement serait particulièrement surprenant :

```
$ ./exemple_scanf_6
Entrez un entier : 12



A
Ok i=12
$
```

Nous avons appuyé trois fois sur la touche « Entrée » à la suite de notre saisie, puis en désespoir de cause, nous avons retapé une lettre quelconque (`A`) suivie de Entrée. Et c'est à ce moment seulement que notre saisie initiale a été validée !

Pourtant ce fonctionnement est tout à fait normal. Comme nous avons mis un '`\n`' en fin de chaîne de format – mais le résultat aurait été le même avec n'importe quel caractère blanc – `scanf()` élimine tous les caractères blancs se trouvant à la suite de notre saisie décimale. Seulement, pour pouvoir éliminer tous les caractères blancs, elle est obligée d'attendre d'en recevoir un qui ne soit pas blanc. Tout ceci explique l'inefficacité de nos multiples pressions sur la touche Entrée, et qu'il ait fallu attendre un caractère non blanc, en l'occurrence A, pour que `scanf()` se termine. Notons que ce caractère non blanc est replacé dans le flux pour la lecture suivante.

Le comportement de `scanf()` est parfois déroutant lorsqu'elle agit directement sur les flux. Pour cela, il est souvent préférable de faire la lecture ligne par ligne grâce à `fgets()` ou à `getline()`, et d'analyser ensuite le résultat avec `sscanf()`. Celle-ci aurait en effet, dans notre dernier exemple, rencontré la fin de la chaîne, qu'elle aurait traitée comme un `EOF`, ce qui lui aurait permis d'arrêter la recherche d'un caractère non blanc. En voici la preuve avec le programme suivant (le test d'erreur sur `fgets()` a été supprimé pour simplifier l'exemple).

exemple_scanf_7.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char ligne[128];
    int  i;

    fprintf(stdout, "Entrez un entier : ");
    fgets(ligne, 128, stdin);
    if (sscanf(ligne, "%d\n", & i) == 1)
        fprintf(stdout, "Ok i=%d\n", i);
    else
        fprintf(stdout, "Erreur \n");
    return EXIT_SUCCESS;
}
```

```
$ ./exemple_scanf_7
Entrez un entier : 12
Ok i=12
$ ./exemple_scanf_7
Entrez un entier : A
Erreur
$
```

Les fonctions de la famille `scanf()` offrent également quelques possibilités moins connues que nous allons voir rapidement.

- La saisie de l'adresse d'un pointeur, avec la directive `%p` : ceci ne doit être utilisé qu'avec une extrême précaution, le programme étant prêt à capturer un signal `SIGSEGV` dès qu'il va essayer de lire le contenu du pointeur si l'utilisateur a fait une erreur.
- La lecture d'un champ sans stockage dans une variable, en insérant un astérisque juste après le caractère `%`. Le champ est purement et simplement ignoré, sans stockage dans un pointeur ni incrémentation du nombre de champs correctement lus. Ceci est surtout utilisé pour ignorer des valeurs lors de la relecture de la sortie d'autre programme. Imaginons par exemple un programme de dessin vectoriel qui affiche les coordonnées X et Y de tous les points qu'il a en mémoire. Lors d'une relecture de ces données, le numéro du point ne présente pas d'intérêt, aussi préfère-t-on l'ignorer avec une lecture du genre :

```
scanf (" point %*d : X = %lf Y = %lf", & x, & y).
```

- La directive `%n` n'effectue pas de conversion mais stocke dans le pointeur correspondant, qui doit être de type `int *`, le nombre de caractères lus jusqu'à présent. Cela peut servir dans l'analyse d'une chaîne contenant plusieurs champs. Supposons par exemple que le premier champ indique de manière numérique le type du champ suivant (0 = entier, 1 = réel). Il est alors commode de stocker la position atteinte après cette première lecture, pour reprendre ensuite l'extraction avec le format approprié dans un second `sscanf()`. En voici une illustration :

exemple_scanf_8.c :

```
    #define _GNU_SOURCE     /* pour avoir getline() */

    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char * ligne;
    int    taille;

    int    position;
    int    type_champ;
    int    entier;
    float  reel;

    while (1) {
        fprintf(stdout, "<type> <valeur> :\n");
        ligne = NULL;
        taille = 0;
        if (getline(& ligne, & taille, stdin) == -1)
            break;
        if (sscanf(ligne, "%d %n", & type_champ, & position) != 1) {
            fprintf(stdout, "Entrez le type (0=int, 1=float) "
                            "suivi de la valeur \n");
            free(ligne);
            continue;
        }
```

```
        if (type_champ == 0) {
            if (sscanf(& (ligne[position]), "%d", &entier) != 1)
                fprintf(stdout, "Valeur entière attendue \n");
            else
                fprintf(stdout, "Ok : %d\n", entier);
        } else if (type_champ == 1) {
            if (sscanf(& (ligne[position]), "%f", & reel) != 1)
                fprintf(stdout, "Valeur réelle attendue \n");
            else
                fprintf(stdout, "Ok : %e\n", reel);
        } else {
            fprintf(stdout, "Type inconnu (0 ou 1)\n");
        }
        free(ligne);
    }
    return EXIT_SUCCESS;
}
```

On arrête la boucle principale de ce programme en faisant échouer `getline()`, en lui envoyant `EOF` (Contrôle-D) en début de ligne. Voici un exemple d'exécution :

```
$ ./exemple_scanf_8
<type> <valeur> :

Entrez le type (0=int, 1=float) suivi de la valeur
<type> <valeur> :
0 A
Valeur entière attendue
<type> <valeur> :
0 12
Ok : 12
<type> <valeur> :
1 Z
Valeur réelle attendue
<type> <valeur> :
1 23.4
Ok : 2.340000e+01
<type> <valeur> :
2 ZZZ
Type inconnu (0 ou 1)
<type> <valeur> :
$
```

Il est également possible de restreindre le jeu de caractères utilisables lors d'une saisie de texte, en utilisant une directive `%[]` à la place de `%s`, et en indiquant à l'intérieur des crochets les caractères autorisés. On peut signaler des intervalles du genre `%[A-Za-z]`, des négations avec le signe `^` en début de directive, comme `%[^0-9]` pour refuser les chiffres. Si on veut mentionner le caractère '`]`', il faut le placer en premier, et pour indiquer '`[`', on le place en dernier, comme dans `%[](){}[]`, qui regroupe les principaux symboles d'encadrement. On notera que cette conversion ne saute pas automatiquement les espaces en tête, contrairement à `%s`. Comme pour cette dernière conversion, il y a lieu d'être prudent pour éviter les débordements de chaînes, soit en mentionnant une taille maximale `%5[A-Z]` qui convertit au plus cinq

majuscules, soit en demandant à la bibliothèque d'allouer la mémoire nécessaire (en lui passant un pointeur sur un pointeur sur une chaîne).

Avec toutes leurs possibilités, les fonctions de la famille `scanf()` sont très puissantes. Toutefois, elles réclament beaucoup d'attention lors de la lecture des données si plusieurs champs sont présents sur la même ligne. Lorsque la syntaxe d'une ligne est très compliquée et qu'une lecture champ par champ comme dans notre dernier exemple est vraiment rébarbative, il est possible de se tourner vers un analyseur syntaxique qu'on pourra construire à l'aide de `flex` et `bison`, par exemple.

Conclusion

Nous avons examiné ici les différentes fonctions d'entrée-sortie simples pour un programme.

Comme nous l'avions déjà indiqué avec `printf()`, l'évolution actuelle des interfaces graphiques conduit les utilisateurs à se détourner des applications dont les données sont saisies depuis un terminal classique. À moins de construire un programme qui, à la manière d'un filtre, reçoive sur son entrée standard des données provenant d'une autre application, il est de plus en plus rare d'utiliser `scanf()` ou `fscanf()` sur `stdin`. Par contre, l'emploi de `sscanf()` est toujours d'actualité. En effet, la saisie de données par l'intermédiaire d'une interface graphique se fait souvent dans une boîte de dialogue, dont les composants de saisie renvoient leur contenu sous forme de chaîne de caractères. Il est alors du ressort du programme appelant de convertir ces chaînes dans le format de donnée qu'il désire (entier, réel, voire pointeur). Il peut utiliser à ce moment `sscanf()` ou d'autres fonctions de conversion que nous étudierons ultérieurement, comme `strtol()`, `strtod()` ou `strtoul()`.

Les commandes de redirection des entrées-sorties standards sont présentées, par exemple, dans [BLAESS 2001] *Langages de scripts sous Linux*. La plupart des fonctions de la bibliothèque C Ansi et principalement `stdio` sont décrites dans [KERNIGHAN 1994] *Le langage C*, qui reste une référence incontournable.

11

Ordonnancement des processus

Dans ce chapitre nous allons approcher les mécanismes sous-jacents lors de l'exécution des programmes. Nous étudierons tout d'abord les différents états dans lesquels un processus peut se trouver, ainsi que l'influence du noyau sur leurs transitions.

Nous analyserons ensuite les méthodes simples permettant de modifier la priorité d'un processus par rapport aux autres.

Enfin, nous observerons les fonctionnalités définies par la norme SUSv3, qui permettent de modifier l'ordonnancement des processus, principalement dans l'esprit d'un fonctionnement temps-réel.

États d'un processus

Indépendamment de toute mécanique d'ordonnancement, un processus peut se trouver dans un certain nombre d'états différents, en fonction de ses activités. Ces états peuvent être examinés à l'aide de la commande `ps` ou en regardant le contenu du pseudo-fichier `/proc/<pid>/status`. Ce dernier contient en effet une ligne `State:...` indiquant l'état du processus. Nous utiliserons de préférence cette seconde méthode, car elle permet d'éviter de lancer un processus supplémentaire (`ps`), qui est le seul à être réellement actif au moment de l'invocation, sur une machine monoprocesseur du moins.

Les différents états d'un processus sont les suivants :

état	Anglais	Signification
Exécution	Running (R)	Le processus est en cours de fonctionnement, il effectue un travail actif.
Sommeil	Sleeping (S)	Le processus est en attente d'un événement extérieur. Il se met en sommeil.
Arrêt	Stopped (T)	Le processus a été temporairement arrêté par un signal. Il ne s'exécute plus et ne réagira qu'à un signal de redémarrage.
Zombie	Zombie (Z)	Le processus s'est terminé, mais son père n'a pas encore lu son code de retour.

Il n'y a pas grand-chose à ajouter sur l'état *Running*. Le processus s'exécute normalement, il a accès au processeur de la machine et avance dans son code exécutable.

Les processus en sommeil sont généralement en attente d'une ressource ou d'un événement extérieur. Dans la plupart des cas, ils attendent le résultat d'une opération d'entrée-sortie. Lorsqu'un programme veut écrire sur son terminal de sortie standard, le noyau prend le contrôle (à travers l'appel-système `write()` sous-jacent) et assure l'écriture. Toutefois, la durée de réaction d'un terminal est relativement longue. Aussi le noyau bascule-t-il le processus en sommeil, en attente de la fin de l'écriture. Une fois que celle-ci est terminée, le processus se réveille et reprend son exécution.

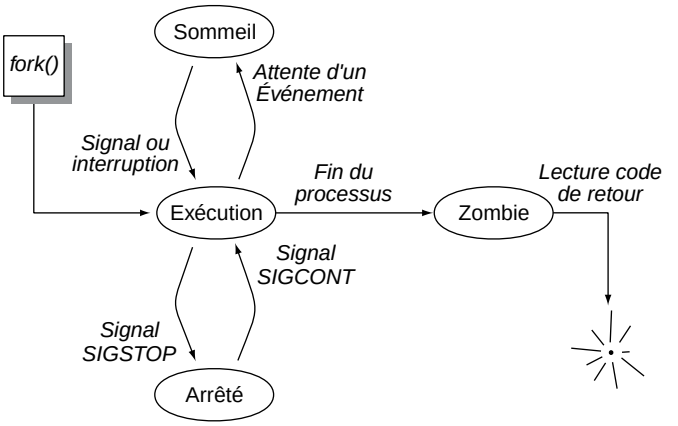
Notons enfin qu'il existe deux types de sommeil : interruptible et ininterruptible. Dans le premier état, un processus peut être réveillé par l'arrivée d'un signal. Dans le second cas, le processus ne peut être réveillé que par une interruption matérielle reçue par le noyau. Le sommeil « ininterruptible » est représenté par la lettre `D` dans le résultat de `ps`. Ce cas est rare mais peut arriver, notamment lorsque des problèmes de montage de systèmes de fichiers (par exemple par protocole *NFS*) entrent en considération. On peut ainsi trouver parfois un processus *mount* bloqué dans un sommeil ininterruptible suite à un problème matériel ; il ne faut pas s'étonner qu'il ne réponde à aucun signal – y compris SIGKILL (9) – et ne disparaîtra pas avant le prochain *reboot* du système.

Lorsqu'un processus reçoit un signal `SIGSTOP`, il est arrêté temporairement mais pas terminé définitivement. Ce signal peut être engendré par l'utilisateur (avec `/bin/kill` par exemple), par le terminal (généralement avec la touche Contrôle-Z), ou encore par un débogueur comme `gdb` qui interrompt le processus pour l'exécuter pas à pas. Le signal `SIGCONT` permet au processus de redémarrer ; il est déclenché soit par `/bin/kill`, soit par le shell (commandes internes `bg` ou `fg`), ou encore par le débogueur pour reprendre l'exécution.

Enfin, un processus qui se termine doit renvoyer une valeur à son père. Cette valeur est celle qui est fournie à l'instruction `return()` de la fonction `main()`ou dans l'argument de la fonction `exit()`. Nous avons déjà abordé ces notions avec les fonctions de la famille `wait()`, étudiées au chapitre 5. Tant que le processus père n'a pas lu cette valeur, le fils reste dans un état intermédiaire entre la vie et la mort, toutes ses ressources ont été libérées, mais il conserve une place dans la table des tâches sur le système. On dit qu'il est à l'état *Zombie*. Si le processus père ignore explicitement le signal `SIGCHLD`, le processus meurt tout de suite, sans rester à l'état zombie. Si, au contraire, le processus père ne lit jamais la valeur de retour et laisse à `SIGCHLD` son comportement par défaut, le fils restera à l'état zombie indéfiniment. Lorsque le processus père se termine à son tour, le fils orphelin est adopté par le processus numéro 1, `init`, qui lit immédiatement sa valeur de retour (même s'il n'en a aucune utilité), permettant enfin à cette pauvre âme d'accéder au repos éternel des processus…

Dans l'exemple suivant, nous allons faire passer un processus – et son fils – par tous ces stades. Tout d'abord, notre processus va consulter son propre état dans le pseudo-système de fichiers `/proc`, puis il va se scinder avec `fork()` avant de s'endormir pendant quelques secondes (en attente donc d'un signal de réveil). Son fils profitera de ce laps de temps pour afficher l'état du père, puis se terminera immédiatement. À son réveil, le processus père examinera l'état de son fils avant et après avoir lu le code de retour. Ensuite, le processus se met en sommeil, en attente d'une saisie de caractères.

Figure 11.1
États successifs d'un processus



exemple_status.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <sys/wait.h>

    void
affiche_status (pid_t pid)
{
    FILE * fp;
    char   chaine[80];
    sprintf(chaine, "/proc/%ld/status", (long) pid);
    if ((fp = fopen(chaine, "r")) == NULL) {
        fprintf(stdout, "Processus inexistant\n");
        return;
    }
    while (fgets(chaine, 80, fp) != NULL)
        if (strncmp(chaine, "State", 5) == 0) {
            fputs(chaine, stdout);
            break;
        }
    fclose(fp);
}

    int
main (void)
{
    pid_t pid;
    char  chaine[5];

    fprintf(stdout, "PID = %ld\n", (long) getpid());
    fprintf(stdout, "État attendu = Running \n");
    affiche_status(getpid());
```

```
    if ((pid = fork()) == -1) {
        perror("fork ()");
        exit(EXIT_FAILURE);
    }
    if (pid != 0) {
        sleep(5);
        fprintf(stdout, "Consultation de l'état de mon fils...\n");
        fprintf(stdout, "État attendu = Zombie \n");
        affiche_status(pid);
        fprintf(stdout, "Père : Lecture code retour du fils...\n");
        wait(NULL);
        fprintf(stdout, "Consultation de l'état de mon fils...\n");
        fprintf(stdout, "État attendu = inexistant\n");
        affiche_status(pid);
    } else {
        fprintf(stdout, "Fils : consultation de l'état du père...\n");
        fprintf(stdout, "État attendu = Sleeping \n");
        affiche_status(getppid());
        fprintf(stdout, "Fils : Je me termine \n");
        exit(EXIT_SUCCESS);
    }
    fprintf(stdout, "Attente de saisie de chaîne \n");
    fgets(chaine, 5, stdin);
    exit(EXIT_SUCCES);
}
```

Nous allons déjà exécuter cet exemple, jusqu'à la saisie de chaîne, en observant les différents états :

```
$ ./exemple_status
PID = 787
État attendu = Running
State:  R (running)
Fils : consultation de l'état du père...
État attendu = Sleeping
State:  S (sleeping)
Fils : Je me termine
Consultation de l'état de mon fils...
État attendu = Zombie
State:  Z (zombie)
Père : Lecture code retour du fils...
Consultation de l'état de mon fils...
État attendu = inexistant
Processus inexistant
Attente de saisie de chaîne
```

Le processus est donc en attente de saisie. Nous allons à ce moment lancer `gdb` depuis une autre console et utiliser la commande `attach` de `gdb` pour déboguer le processus en cours de fonctionnement. Nous pourrons alors lancer la commande `cat /proc/787/status` à l'aide de l'instruction `shell` de `gdb` :

```
$ gdb exemple_status
GNU gdb 4.18
```

```
Copyright 1998 Free Software Foundation, Inc.
  [...]
 (gdb) attach 787
Attaching to program: /home/ccb/Exemples/exemple_status, Pid 787
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
0x400bddb4 in __libc_read () from /lib/libc.so.6
(gdb) shell cat /proc/787/status | grep State
State:  T (stopped)
(gdb) detach
Detaching from program: /home/ccb/Exemples/exemple_status, Pid 787
(gdb) quit
$
```

Cela nous permet de vérifier que le processus en cours de débogage est bien arrêté entre les exécutions pas à pas.

Fonctionnement multitâche, priorités

Jusqu'à présent, nous avons considéré que le processus est seul, qu'il s'exécute sur un processeur qui lui est attribué. Cela peut être le cas sur une machine multiprocesseur (SMP) peu chargée, mais c'est rare. Sur un système multitâche, l'accès au processeur est une ressource comme les autres (mémoire, disque, terminaux, imprimantes…) qui doit se partager entre les processus concurrents.

Dans le cadre de ce chapitre, nous considérerons principalement le cas des machines uniprocesseurs. On pourra presque toujours généraliser notre propos aux ordinateurs multiprocesseurs dès que le nombre de processus concurrents sur le système augmentera.

Le noyau est chargé d'assurer la commutation entre les différents programmes pour leur offrir l'accès au processeur. C'est très facile lorsqu'un processus s'endort. N'oublions pas que toutes les transitions entre états que nous avons vues plus haut se passent toujours par l'intermédiaire d'un appel-système ou de l'arrivée d'un signal. Aussi, le noyau a toujours le contrôle total de l'état des processus. Lorsque l'un d'eux s'endort, en attente d'une saisie de l'opérateur par exemple, le noyau peut alors élire un autre programme pour lui attribuer l'accès au processeur.

Seulement ce n'est pas encore suffisant, on ne peut guère compter sur la bonne volonté de chaque processus pour s'endormir régulièrement pour laisser un peu de temps CPU disponible pour les autres. La moindre erreur de programmation du genre

```
    while (1);
        /* Exécuter un travail */
```

bloquerait totalement le système (notez le « ; » en trop à la suite du `while()`, qui engendre une boucle infinie).

Pour éviter ce problème, le noyau doit interrompre au bout d'un certain temps un processus qui s'exécute sans avoir besoin de s'endormir (en effectuant des calculs par exemple), afin qu'il cède la place à un autre programme. On dit que le noyau réalise une **préemption** du processeur, ce qui a donné naissance au terme de multitâche préemptif.

Avec ce mécanisme, le noyau peut intervenir lorsqu'un processus dépasse le quantum de temps qui lui est imparti. Le processus passe alors à l'état *Prêt*. Un processus prêt est donc

simplement un processus en cours d'exécution qui a été suspendu par le noyau et qui reprendra son travail dès que celui-ci lui réaffectera le processeur.

Au niveau de la commande `ps` ou du pseudo-fichier `/proc/<pid>/status`, il n'y a aucune différence entre un processus effectivement en exécution et un processus prêt. Ils sont tous deux indiqués par la lettre `R`. Cela explique pourquoi la commande `ps aux` présente parfois une liste contenant simultanément plusieurs processus à l'état `R` sur une machine uni-processeur.

Un corollaire de ce mécanisme est qu'un processus qui se réveille, par exemple à cause d'une opération d'entrée-sortie terminée ou de l'arrivée d'un signal, ne passe pas directement de l'état Sommeil à l'état Exécution, mais passe par un état transitoire Prêt, et c'est l'ordonnanceur qui décidera du véritable passage en Exécution.

Une application n'a donc habituellement aucune raison de se soucier de l'ordonnancement assuré par le noyau, tout se passe de manière totalement transparente. Un processus a toujours l'impression d'être en cours d'exécution. Malgré tout, il faut être conscient qu'une application qui effectue de larges plages d'opérations sans réclamer d'entrée-sortie emploie beaucoup la seule ressource qui soit vraiment indispensable pour tous les processus : le CPU. Cette application pénalisera donc les autres logiciels qui font un usage plus raisonnable du processeur.

Le noyau utilise, pour pallier ce problème, le principe de priorité double. Une valeur statique de priorité est donnée à tout processus dès son démarrage et peut partiellement être corrigée par un appel-système approprié. L'ordonnanceur se sert de cette valeur statique pour calculer une nouvelle valeur, nommée priorité dynamique, et qui est remise à jour à chaque fois que le processus est traité par l'ordonnanceur. Cette priorité dynamique est entièrement interne au noyau et ne peut pas être modifiée. Plus un processus utilise le temps CPU qui lui est imparti, plus le noyau diminue sa priorité dynamique. Au contraire, plus le processus rend vite la main lorsqu'on l'exécute, plus le noyau augmente sa priorité. Avec cette politique, les tâches qui exploitent peu le processeur – déclenchant une opération d'entrée-sortie et s'endormant aussitôt en attente du résultat – passeront beaucoup plus rapidement de l'état Prêt à l'Exécution effective que les tâches qui grignotent tous les cycles CPU qu'on leur offre, sans jamais redonner la main volontairement.

Cette organisation de l'ordonnanceur permet d'améliorer en partie la fluidité du système. Malgré tout, lorsqu'on lance sur une machine uni-processeur plusieurs processus qui dévorent les cycles CPU en boucle, une nette diminution des performances du système est sensible. Pourtant, une grande partie des programmes qui font beaucoup de calculs et peu d'entrées-sorties ne présentent pas de caractère d'urgence. En voici quelques exemples :

- Une application reçoit des informations numériques, les affiche et les rediffuse. Elle exécute régulièrement des opérations de calcul statistique dont le résultat n'est imprimé qu'une fois par mois.
- Un système d'enregistrement collecte des blocs de données, les regroupe en petits fichiers correspondant chacun à une heure de trafic, puis les transfère sur un répertoire accessible en FTP anonyme, où d'autres machines viendront les récupérer. Pour diminuer le volume des transferts ultérieurs, on comprime les fichiers en invoquant `gzip` par exemple.
- Une application de création d'image numérique permet d'utiliser un modeleur affichant la scène en préparation sous forme de squelette adapté à la définition de l'écran. Avant le transfert vers le système de photocomposition définitif, l'image en très haute résolution est calculée par un processus complexe de calcul (*ray-tracing* par exemple).

Dans tous ces exemples, on remarque qu'une partie du travail est fortement consommatrice de cycles CPU (le calcul statistique, la compression de données, le traitement d'image) alors que le résultat n'est pas indispensable immédiatement. On ne peut toutefois pas se reposer entièrement sur le noyau pour « freiner » systématiquement ce genre d'opérations, car il existe aussi de nombreux cas où on attend impatiemment le résultat d'un travail intense du processeur. Le meilleur exemple pour le développeur est probablement la recompilation d'un logiciel relativement conséquent, et la surdose de caféine et de sucreries qui finit par meubler l'attente obligatoire…

Il est donc nécessaire de pouvoir donner au noyau une indication de la priorité qu'on affecte à tel ou tel travail. Lorsque plusieurs processus seront prêts, le noyau choisira d'abord celui dont la priorité dynamique est la plus importante. Lors du calcul de la priorité dynamique, l'ordonnanceur utilise la priorité statique conjointement à d'autres facteurs, comme le fait que le processus ait relâché le processeur avant l'expiration de son délai, l'emplacement réel du processus – sur les systèmes multiprocesseurs – ou la disponibilité immédiate de son espace d'adressage complet (ce qui concerne principalement les threads que nous étudierons au prochain chapitre).

Plus un processus a une priorité dynamique élevée, plus la tranche de temps qu'on lui allouera sera longue. C'est un moyen de punir les programmes qui bouclent, en les laissant travailler quand même, mais sans trop perturber les autres processus. Lorsqu'un processus a consommé tous ses cycles, il ne sera réélu pour l'accès au processeur que dans le cas où aucun autre processus plus courtois n'est prêt.

Lorsqu'un processus désire influer sur sa propre priorité statique, il peut utiliser l'appel-système `nice()`. On indique à celui-ci la « gentillesse » dont le processus appelant désire faire preuve. La déclaration de cette fonction se trouve dans `<unistd.h>` :

```
int nice (int increment) ;
```

La valeur transmise est ajoutée à notre gentillesse vis-à-vis des autres processus. Cela signifie qu'un incrément positif diminue la priorité du processus, alors qu'un incrément négatif augmente sa priorité. Seul un processus ayant l'UID effectif de *root* ou la capacité `CAP_SYS_NICE` peut diminuer sa gentillesse. La plage de valeur utilisée en interne par l'ordonnanceur pour les priorités statiques s'étale de 0 à 40. Toutefois, par convention on présente la gentillesse d'un processus sur une échelle allant de –20 (processus très égoïste) à +20, la valeur 0 étant celle par défaut. Un utilisateur normal ne peut donc avoir accès qu'à la plage allant de 0 à 20.

Dans l'exemple suivant, le programme va lancer cinq processus fils, chacun d'eux prenant une valeur de gentillesse différente et se mettant à boucler sur un comptage. Le processus père les laisse travailler pendant 5 secondes et les arrête. Pour synchroniser le début et la fin du comptage pour les différents fils, nous utilisons un signal émis par le père à destination du groupe de processus. Pour être sûr que les fils ont tous démarré avant d'envoyer le signal, le père respecte un petit sommeil d'une seconde ; c'est une méthode imparfaite, mais qui fonctionne sur un système peu chargé.

exemple_nice.c :

```
#define _GNU_SOURCE
#include <signal.h>
#include <stdio.h>
```

```
    #include <stdlib.h>
    #include <sys/wait.h>
    #include <unistd.h>

    volatile long compteur = 0;
    static int   gentillesse;

    void
gestionnaire (int numero)
{
    if (compteur != 0) {
        fprintf(stdout, "Fils %ld (nice %2d) Compteur = %9ld\n",
                          (long) getpid(), gentillesse, compteur);
        exit(EXIT_SUCCESS);
    }
}

    #define NB_FILS    5

    int
main (void)
{
    pid_t   pid;
    int     fils;

    /* Création d'un nouveau groupe de processus */
    setpgid(0, 0);

    signal(SIGUSR1, gestionnaire);
    for (fils = 0; fils < NB_FILS; fils ++) {
        if ((pid = fork()) < 0) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (pid != 0)
            continue;
        gentillesse = fils * (20 / (NB_FILS - 1));
        if (nice(gentillesse) < 0) {
            perror("nice");
            exit(EXIT_FAILURE);
        }
        /* attente signal de démarrage */
        pause();
        /* comptage */
        while (1)
            compteur ++;
    }
    /* processus père */
    signal(SIGUSR1, SIG_IGN);
    sleep(1);
    kill(-getpgid(0), SIGUSR1);
```

```
    sleep(5);
    kill(-getpgid(0), SIGUSR1);
    while (wait(NULL) > 0)
        ;
    exit(EXIT_SUCCESS);
}
```

L'exécution montre bien que les processus ont eu un accès différent au processeur, inversement proportionnel à leur indice de gentillesse :

```
$ ./exemple_nice
Fils 1849 (nice 10) Compteur =  91829986
Fils 1850 (nice 15) Compteur =  42221693
Fils 1851 (nice 20) Compteur =  30313573
Fils 1847 (nice  0) Compteur = 183198223
Fils 1848 (nice  5) Compteur = 133284576
$
```

Cela intéressera donc le programmeur dont l'application déclenche plusieurs processus, certains effectuant beaucoup de calculs peu urgents. De telles tâches auront intérêt à diminuer leur priorité (augmenter leur gentillesse) pour conserver un fonctionnement plus fluide au système.

Modification de la priorité d'un autre processus

Pouvoir modifier sa propre priorité est une bonne chose, mais il y a de nombreux cas où on aimerait changer la priorité d'un autre processus déjà en exécution. L'exemple est fourni par l'utilitaire `/usr/bin/renice`, qui permet de diminuer la priorité d'un processus s'il charge trop lourdement le processeur ou au contraire de l'augmenter (à condition d'avoir les privilèges nécessaires). Ces opérations sont possibles grâce à deux appels-système **`getpriority()`** et **`setpriority()`** qui sont déclarés dans `<sys/wait.h>` :

```
int getpriority (int classe, int identifiant)
int setpriority (int classe, int identifiant, int valeur)
```

Ces deux appels-système ne travaillent pas obligatoirement sur un processus particulier, mais peuvent agir sur un groupe de processus ou sur tous les processus appartenant à un utilisateur donné.

En fonction de la classe indiquée en premier argument, l'identifiant fourni en second est interprété différemment :

Valeur de classe	Type d'identifiant
`PRIO_PROCESS`	PID du processus visé.
`PRIO_PGRP`	PGID du groupe de processus concerné.
`PRIO_USER`	UID de l'utilisateur dont on vise tous les processus.

La valeur de retour de `getpriority()` correspond à la priorité statique du processus, qui s'étend dans l'intervalle PRIO_MIN à PRIO_MAX, qui valent typiquement –20 et 20. Aussi lorsque `getpriority()` renvoie –1, on ne peut être sûr qu'il s'agit d'une erreur qu'à condition

d'avoir mis `errno` à 0 avant l'appel, et en vérifiant alors son état. Lorsque plusieurs processus sont concernés par `getpriority()`, la valeur renvoyée se rapporte à la plus petite de toutes leurs priorités. Les valeurs de priorité considérées ici représentent en réalité des quantités de gentillesse. Plus la valeur est petite (proche de –20), plus le processus est prioritaire. L'exemple suivant va nous permettre de consulter les priorités.

exemple_getpriority.c :

```
    #include <ctype.h>
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/resource.h>

    void
syntaxe (char * nom)
{
    fprintf(stderr, "Syntaxe : %s <classe> <identifiant> \n", nom);
    fprintf(stderr, "   <classe> = P (PID)\n");
    fprintf(stderr, "              G (PGID)\n");
    fprintf(stderr, "              U (UID)\n");
    exit(EXIT_FAILURE);
}

    int
main (int argc, char * argv[])
{
    int  classe;
    int  identifiant;
    int  priorite;

    if (argc != 3)
        syntaxe(argv[0]);
    if (toupper(argv[1][0]) == 'P')
        classe = PRIO_PROCESS;
    else if (toupper(argv[1][0]) == 'G')
        classe = PRIO_PGRP;
    else if (toupper(argv[1][0]) == 'U')
        classe = PRIO_USER;
    else
        syntaxe(argv[0]);
    if (sscanf(argv[2], "%d", & identifiant) != 1)
        syntaxe(argv[0]);
    errno = 0;
    priorite = getpriority(classe, identifiant);
    if ((priorite == -1) && (errno != 0))
        perror(argv[2]);
    else
        fprintf(stderr, "%d : %d\n", identifiant, priorite);
    return EXIT_SUCCESS;
}
```

L'exemple d'exécution ci-dessous présente un intérêt limité. Il est plus utile de comparer les valeurs obtenues par ce programme et celles qu'on visualise avec la commande `ps`, ou encore mieux avec le logiciel `top`.

```
$ ./exemple_getpriority P 2982
2982 : 15
$ ./exemple_getpriority U 500
500 : 0
$ ./exemple_getpriority P 6
6 : -20
$
```

`setpriority()` fonctionne de manière symétrique, en fixant la nouvelle priorité statique du ou des processus indiqués. Bien sûr, des restrictions s'appliquent en ce qui concerne les droits d'accès au processus, et seul *root* (ou un programme ayant le privilège `CAP_SYS_NICE`) peut rendre un processus plus prioritaire.

Les fonctions `nice()`, `getpriority()` ou `setpriority()` ne sont pas définies par Ansi C (qui n'inclut pas le concept de multitâche) mais elles sont indiquées dans SUSv3. Toutefois, si ces appels-système peuvent suffire pour de petites opérations de configuration administrative (accélérer un calcul par rapport à un autre, ou diminuer la priorité des jobs ne présentant pas de caractère d'urgence), ils sont largement insuffisants dès qu'on a réellement besoin de configurer le comportement de l'ordonnanceur en détail.

Pour cela, plusieurs types d'ordonnancements ont été définis dans la norme Posix.1b, offrant ainsi des possibilités approchant de la véritable programmation temps-réel.

Les mécanismes d'ordonnancement sous Linux

Les fonctionnalités décrites à l'origine par la norme Posix.1b, sont disponibles à condition que la constante symbolique **`_POSIX_PRIORITY_SCHEDULING`** soit définie par inclusion du fichier d'en-tête `<unistd.h>`. Elles sont alors déclarées dans le fichier `<sched.h>`. On pourra donc utiliser un code du genre

```
    #include <unistd.h>
    #ifdef _POSIX_PRIORITY_SCHEDULING
        #include <sched.h>
    #else
        #warning "Fonctionnalités temps-réel non disponibles"
    #endif
```

et dans le corps du programme

```
    int
main(void)
{
    #ifdef _POSIX_PRIORITY_SCHEDULING
        /* Basculement en ordonnancement temps-réel */
    #else
        /* Utilisation uniquement de nice() */
    #endif
    ...
}
```

Il existe trois types d'ordonnancement. En fait, il n'y a qu'un seul ordonnanceur, mais il choisit ou il rejette les processus selon trois politiques possibles. La configuration se fait processus par processus. Elle n'est pas nécessairement globale. Par contre, la modification de la politique d'ordonnancement associée à un processus est une opération privilégiée, car il existe un – gros – risque de bloquer complètement le système.

Les trois algorithmes d'ordonnancement sont nommés FIFO, RR et OTHER.

Ordonnancement sous algorithme FIFO

L'algorithme FIFO est celui d'une file d'attente (*First In First Out* – premier arrivé, premier servi).

Dans cette optique, il existe une liste des processus pour chaque priorité statique. Le premier processus de la priorité la plus haute s'exécute jusqu'à ce qu'il relâche le processeur. Il est alors repoussé à la fin de la liste correspondant à sa priorité. Si un autre processus de même priorité est prêt, il est élu. Sinon, l'ordonnanceur passe au niveau de priorité inférieur et en extrait le premier processus prêt. Ce mécanisme se répète indéfiniment. Dès qu'un processus de priorité supérieure à celui qui est en cours d'exécution est de nouveau prêt (parce qu'il attendait une entrée-sortie qui vient de se terminer par exemple), l'ordonnancement lui attribue immédiatement le processeur.

Il existe un appel-système particulier, `sched_yield()`, qui permet à un processus en cours d'exécution de relâcher volontairement le processeur. L'ordonnanceur peut alors faire tourner un autre processus du même niveau de priorité, s'il y en a un qui est prêt. Par contre, si aucun autre processus de même niveau n'est prêt à s'exécuter, le processeur est réattribué au processus qui vient d'invoquer `sched_yield()`. Le noyau n'élit jamais un processus si un autre de priorité supérieure est prêt.

Cet ordonnancement est le plus violent et le plus égoïste qui soit. Le plus fort a toujours raison. Le processus le plus prioritaire a toujours le processeur dès qu'il est prêt à s'exécuter. Il existe bien entendu un très sérieux risque de blocage du système (du moins sur une machine uni-processeur) si on exécute un simple

```
while (1)
    ;
```

avec une priorité élevée.

Pour éviter ce genre de désagrément lors d'une phase de débogage, il est important de conserver un shell s'exécutant à un niveau de priorité plus élevé que le processus en cours de développement. On pourra alors exécuter facilement un `kill -KILL...`

Si on travaille dans l'environnement X-Window, un shell de niveau supérieur ne suffit pas ; nous verrons qu'il faut aussi faire fonctionner le serveur X et tout son environnement avec une priorité plus grande que le processus en débogage. Dans ce cas en effet, il ne suffit pas d'avoir un Xterm pour arrêter le processus fautif, mais encore faut-il que le serveur X soit capable de faire bouger le pointeur de la souris jusqu'à la fenêtre du Xterm de secours, et que le gestionnaire de fenêtre arrive à activer cette dernière. Nous verrons des exemples de blocages volontaires (et temporaires) du système dans les prochaines sections.

Un programme se trouvant seul au niveau de priorité FIFO le plus élevé est sûr de s'exécuter de bout en bout sans être perturbé. Par contre, si deux processus s'exécutent au même niveau FIFO, la progression parallèle des deux n'est pas très prévisible. La commutation s'effectue

parfois volontairement, en invoquant `sched_yield()`, et parfois sur des appels-système bloquants qui endorment un processus.

Le comportement d'un processus seul est presque totalement déterministe (aux retards induits par les interruptions matérielles près). Cela permet d'assurer un comportement temps-réel quasi parfait. Par contre, deux processus concurrents à la même priorité ont des progressions imprévisibles. Pour améliorer tout cela, un second type d'ordonnancement temps-réel a été défini.

Ordonnancement sous algorithme RR

L'ordonnancement RR (*Round Robin*, tourniquet) est une simple variante de celui qui a été décrit précédemment, incorporant un aspect préemptif au temps partagé. Chaque processus se voit attribuer une tranche de temps fixe. Lorsqu'il a atteint sa limite, le noyau l'interrompt et le met en état Prêt. Ensuite, il le repousse à la fin de la liste des processus associée à sa priorité. Si un autre processus du même niveau est prêt, il sera choisi. Si aucun autre processus de même priorité n'est prêt, le noyau redonne la main au programme qu'il vient d'interrompre. On ne donne jamais le processeur à un processus de plus faible priorité.

La différence avec l'algorithme FIFO réside donc uniquement dans le cas où plusieurs processus sont simultanément prêts avec la même priorité (et si aucun processus de plus haute priorité n'est prêt). Dans le cas de l'algorithme FIFO, le premier processus qui arrive reçoit le processeur et le conserve jusqu'à ce qu'il s'endorme ou qu'il le relâche volontairement avec `sched_yield()`. Avec l'ordonnancement RR, chaque processus prêt de la plus haute priorité sera régulièrement choisi pour s'exécuter, quitte à interrompre l'un de ses confrères qui ne veut pas s'arrêter de lui-même.

Si deux processus ont la même priorité, chacun aura donc l'impression de s'exécuter deux fois moins vite que s'il était seul, mais aucun des deux ne sera bloqué pour une période *a priori* inconnue, comme c'était le cas avec l'ordonnancement FIFO.

Ordonnancement sous algorithme OTHER

Le troisième type d'ordonnancement est l'algorithme OTHER (autre), qui n'est pas réellement défini par Posix.1b. L'implémentation de cet algorithme est laissée à la discrétion des concepteurs du noyau. Sur certains systèmes, il peut s'agir d'ailleurs de l'algorithme RR, avec des plages de priorité plus faibles.

Sous Linux, il s'agit de l'ordonnancement par défaut dont nous avons déjà parlé, utilisant une priorité dynamique recalculée en fonction de la priorité statique et de l'usage que le processus fait du laps de temps qui lui est imparti.

Il est important de savoir que les algorithmes dits temps-réel (FIFO et RR) ont des plages de priorité qui sont toujours supérieures à celles des processus s'exécutant avec l'algorithme OTHER. Autrement dit, un processus FIFO ou RR aura toujours la préséance sur tous les processus OTHER, même ceux dont la gentillesse est la moindre.

Récapitulation

L'ordonnanceur fonctionne donc ainsi :

1. S'il existe un ou plusieurs processus FIFO ou RR prêts, ils sont sélectionnés en premier. Celui dont la priorité est la plus grande est choisi. S'il s'agit d'un processus RR, on

programme un délai au bout duquel le processus sera rejeté en fin de sa liste de priorité s'il n'a pas rendu le processeur auparavant.

2. Si aucun processus FIFO ou RR n'est prêt, le noyau recalcule les priorités dynamiques des processus OTHER prêts, en fonction de leurs priorités statiques, de leur utilisation du CPU, et d'autres paramètres (emplacement sur une machine multiprocesseur, disponibilité de l'espace mémoire…). En fonction de la priorité dynamique, un processus est élu, et le noyau lui attribue le processeur pendant un délai maximal.
3. Si aucun processus n'est prêt, le noyau peut arrêter le processeur sur les architectures i386 par exemple, jusqu'à l'arrivée d'une interruption signalant un changement d'état.

Temps-réel ?

Nous avons évoqué à plusieurs reprises le terme d'ordonnancement temps-réel. Certains sont sceptiques, et à juste titre, sur l'emploi de ce mot à propos de Linux ou de tout système Unix en général.

Il existe deux classes de problèmes relevant de la programmation temps-réel :

- Le temps-réel strict, ou dur, impose pour chaque opération des délais totalement infranchissables, sous peine de voir des événements catastrophiques se produire. Il s'agit par exemple du contrôle de la mise à feu d'un réacteur d'avion, du déclenchement d'un Airbag, ou de l'émission des impulsions laser en microchirurgie. La sensibilité par rapport à la limite temporelle est telle qu'il est non seulement impensable de soumettre le processus aux retards dus à d'autres processus, mais également impossible d'admettre la moindre tolérance par rapport au travail même du noyau. L'arrivée d'une interruption devant faire basculer un processus à l'état Prêt, la vérification des tâches en cours, pour finalement laisser la main au même processus, peut induire un retard critique dans ces systèmes. Pour ce type d'application, Linux n'est pas approprié. Il est dans ce cas indispensable de se tourner vers d'autres systèmes d'exploitation spécialisés dans le temps-réel strict, voire le projet RTAI dont le principe est de faire tourner Linux comme une tâche d'un micro-noyau temps-réel.
- Les applications temps-réel souples (*soft*) n'ont pas de contraintes aussi strictes que les précédentes. Les limites temporelles existent toujours, mais les conséquences d'un dépassement faible ne sont pas aussi catastrophiques. Dans ce genre d'application, il est important de subdiviser le système en sous-unités réalisant des tâches bien précises, et dont les priorités peuvent être fixées avec précision. Une application pourra par exemple privilégier la réception et le décodage de données provenant de divers équipements. La transmission des alarmes sur défaut sera probablement traitée aussi avec une haute importance, tandis que l'affichage continu à destination d'un opérateur pourra être abordé avec une priorité légèrement plus faible (si un retard de présentation n'est pas critique). Enfin, on emploiera une priorité radicalement moindre pour des tâches administratives de statistiques, d'impression de copies d'écran ou de journalisation des changements d'état.

En utilisant un ordonnancement RR, voire FIFO, Linux peut être parfaitement adapté à des applications du domaine temps-réel souple. Le comportement est déterministe entre les processus. Un programme de plus faible priorité ne viendra jamais perturber un processus de haute priorité. Les seuls écarts temporels possibles sont dus à la gestion interne du noyau, qui est optimisée, et n'induit que des retards infimes.

Notons que les fonctionnalités temps-réel de Linux ont été améliorées sensiblement dans le noyau 2.6 : l'ordonnanceur lui-même a été modifié pour qu'il réalise son travail (trouver le bon processus parmi ceux en attente et lui donner le CPU) en un temps constant quel que soit le nombre de processus en attente sur le système. Auparavant, pour décider quel est le processus le plus adapté à un moment donné, le noyau prenait un temps proportionnel au nombre de processus prêts. À présent ce temps est fixe.

De plus le noyau lui-même est devenu *préemptible* (qui peut être préempté), à ne pas confondre avec *préemptif* (qui peut préempter une tâche, ce qui était déjà le cas). Ceci signifie que l'exécution d'un appel système n'est plus nécessairement atomique, et qu'un processus peut se trouver interrompu même lorsqu'il exécute du code appartenant au noyau.

Pour un bon exemple de l'utilisation des ordonnancements temps-réel, on peut considérer les applications `cdda2wav` et `cdrecord`, qui permettent sous Linux, respectivement d'extraire des pistes audio d'un CD pour créer des fichiers au format `.WAV`, et de graver un CD à partir de pistes audio ou d'images ISO-9660 d'une arborescence de fichiers. Ces deux utilitaires, lorsqu'ils sont exécutés avec l'UID effectif de *root*, basculent sur un ordonnancement temps-réel. Lorsque `cdda2wav` extrait des données audio, il doit rester en parfaite synchronisation avec le flux de bits qui lui sont transmis (le format audio des CD ne permet pas de contenir des informations de contrôle des données). Il s'exécute donc avec une priorité supérieure à celle de tous les autres processus classiques. Cette application faisant surtout des opérations de lecture-écriture, elle ne ralentit pourtant que très peu les autres programmes. Par contre, `cdrecord` – du moins lorsqu'il est connecté à un graveur sur port parallèle – doit assurer un débit particulièrement constant des données, ce qui nécessite des phases d'attente active (*polling*) au cours desquelles les autres processus sont plus fortement pénalisés.

Modification de la politique d'ordonnancement

La politique d'ordonnancement est héritée au travers d'un `fork()` ou d'un `exec()`. Il est donc possible pour un processus de modifier sa propre politique, puis de lancer un shell afin d'expérimenter les différents ordonnancements. Pour modifier son ordonnancement, un processus doit avoir la capacité `CAP_SYS_NICE`, aussi allons-nous créer un programme que nous installerons Set-UID *root*, permettant de lancer une commande avec l'ordonnancement RR à la priorité voulue. Pour éviter les problèmes de sécurité, ce programme reprendra l'identité de l'utilisateur qui l'a lancé avant d'exécuter la commande voulue.

Les sources habituelles d'informations traitant des processus (`ps`, `top`, `/proc/<pid>/...`) ne nous indiquent pas la politique d'ordonnancement avec laquelle s'exécute un programme. Nous allons donc créer un petit programme qui va nous servir de frontal pour l'appel-système **`sched_getscheduler()`**. Tous les appels-système que nous allons étudier ici sont définis par la norme Posix.1b et sont déclarés dans `<sched.h>` :

```
int sched_getscheduler (int pid);
```

Cet appel-système renvoie –1 en cas d'erreur, sinon il renvoie l'une des trois constantes **`SCHED_FIFO`**, **`SCHED_RR`** ou **`SCHED_OTHER`**, en fonction de l'ordonnancement du processus dont on fournit le PID. Si on passe un PID nul, cette fonction renvoie la politique du processus appelant.

exemple_getscheduler.c

```
    #include <sched.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    void
syntaxe (char * nom)
{
    fprintf(stderr, "Syntaxe %s Pid \n", nom);
    exit(EXIT_FAILURE);
}

    int
main (int argc, char * argv[])
{
    int ordonnancement;
    int pid;

    if ((argc != 2) || (sscanf(argv[1], "%d", & pid) != 1))
        syntaxe(argv[0]);
    if ((ordonnancement = sched_getscheduler(pid)) < 0) {
        perror("sched_getscheduler");
        exit(EXIT_FAILURE);
    }
    switch (ordonnancement) {
        case SCHED_RR :    fprintf(stdout, "RR \n");    break;
        case SCHED_FIFO :  fprintf(stdout, "FIFO\n");  break;
        case SCHED_OTHER : fprintf(stdout, "OTHER\n"); break;
        default :          fprintf(stdout, "???\n");   break;
    }
    return EXIT_SUCCESS;
}
```

L'exécution permet de vérifier que les processus courants s'exécutent sous l'ordonnancement OTHER. Nous réutiliserons ce programme lorsque nous aurons modifié notre propre politique.

```
$ ps
  PID TTY          TIME CMD
  693 pts/0    00:00:00 bash
  790 pts/0    00:00:00 ps
$ ./exemple_getscheduler 693
OTHER
$ ./exemple_getscheduler 0
OTHER
$ ./exemple_getscheduler 1
OTHER
$
```

Nous avons mentionné que les processus temps-réel disposaient d'une priorité statique toujours supérieure à celle des processus classiques, mais les intervalles ne sont pas figés suivant les

systèmes. Il est important, pour respecter la portabilité d'un programme, d'utiliser les appels-système **sched_get_priority_max()** et **sched_get_priority_min()**, qui donnent les valeurs minimales et maximales des priorités associées à une politique d'ordonnancement donnée.

```
int sched_get_priority_min (int politique);
int sched_get_priority_max (int politique);
```

Leur emploi est évident.

exemple_get_priority.c :

```
    #include <sched.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    fprintf(stdout, "Ordonnancement FIFO :\n  %d <= priorité <= %d\n",
            sched_get_priority_min(SCHED_FIFO),
            sched_get_priority_max(SCHED_FIFO));
    fprintf(stdout, "Ordonnancement RR :\n  %d <= priorité <= %d\n",
            sched_get_priority_min(SCHED_RR),
            sched_get_priority_max(SCHED_RR));
    fprintf(stdout, "Ordonnancement OTHER :\n  %d <= priorité <= %d\n",
            sched_get_priority_min(SCHED_OTHER),
            sched_get_priority_max(SCHED_OTHER));
    return EXIT_SUCCESS;
}
```

Les intervalles, sous Linux, sont les suivants :

```
$ ./exemple_get_priority
Ordonnancement FIFO :
  1 <= priorité <= 99
Ordonnancement RR :
  1 <= priorité <= 99
Ordonnancement OTHER :
  0 <= priorité <= 0
$
```

Nous voyons que la priorité statique d'un processus OTHER est toujours nulle, mais qu'elle est pondérée par la valeur de gentillesse (étudiée plus haut) afin de permettre de calculer la priorité dynamique.

Lorsqu'une application doit attribuer des priorités différentes à ses processus, exécutés sur un modèle RR ou FIFO, elle doit d'abord consulter les valeurs fournies par ces deux appels-système, et échelonner ses priorités dans l'intervalle disponible. C'est la seule manière d'être vraiment portable selon SUSv3.

Pour connaître ou modifier la priorité statique d'un processus, il faut utiliser une structure de type **sched_param**. Celle-ci peut contenir divers champs, mais le seul qui soit défini par Posix.1b est **sched_priority**, qui représente bien entendu la priorité statique du processus.

Les deux appels-système **sched_getparam()** et **sched_setparam()** permettent de modifier le paramétrage d'un processus.

```
int sched_getparam (pid_t pid, struct sched_param * param);
int sched_setparam (pid_t pid, const struct sched_param * param);
```

L'un comme l'autre renvoient 0 si la consultation ou la modification a réussi, et –1 en cas d'échec.

Il existe un autre paramètre, consultable mais non modifiable : il s'agit de la durée de la tranche de temps affectée à un processus lorsqu'il est ordonnancé sur le mode RR. L'appel-système **sched_rr_get_interval()** permet de lire cette durée.

```
int sched_rr_get_interval (pid_t pid, struct timespec * intervalle);
```

La structure `timespec` que nous avons déjà rencontrée propose les champs `tv_sec`, qui représente des secondes, et `tv_nsec`, qui contient le complément en nanosecondes.

Nous pouvons donc compléter notre programme pour lire toutes les informations concernant l'ordonnancement d'un processus.

exemple_ordonnancement.c :

```
    #include <errno.h>
    #include <sched.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/time.h>
    #include <sys/resource.h>

    void
syntaxe (char * nom)
{
    fprintf(stderr, "Syntaxe %s Pid \n", nom);
    exit(EXIT_FAILURE);
}

    int
main (int argc, char * argv[])
{
    int                 pid;
    int                 ordonnancement;
    int                 prior;
    struct sched_param  param;
    struct timespec     intervalle;

    if ((argc != 2) || (sscanf(argv[1], "%d", & pid) != 1))
        syntaxe(argv[0]);
    if (pid == 0)
        pid = (int) getpid();
    if ((ordonnancement = sched_getscheduler(pid)) < 0) {
        perror("sched_getscheduler");
        exit(EXIT_FAILURE);
    }
```

```
    if (sched_getparam(pid, & param) < 0) {
        perror("sched_getparam");
        exit(EXIT_FAILURE);
    }
    if (ordonnancement == SCHED_RR)
        if (sched_rr_get_interval(pid, & intervalle) < 0) {
            perror("sched_rr_get_interval");
            exit(EXIT_FAILURE);
        }
    if (ordonnancement == SCHED_OTHER) {
        errno = 0;
        if (((prior = getpriority(PRIO_PROCESS, pid)) == -1)
         && (errno != 0)) {
            perror("getpriority");
            exit(EXIT_FAILURE);
        }
    }
    switch (ordonnancement) {
        case SCHED_RR :
            printf("RR : Priorité = %d, intervalle = %ld.%09ld s. \n",
                param.sched_priority,
                intervalle.tv_sec,
                intervalle.tv_nsec);
            break;
        case SCHED_FIFO :
            printf("FIFO : Priorité = %d \n",
                param.sched_priority);
                break;
        case SCHED_OTHER :
            printf("OTHER : Priorité statique = %d dynamique = %d \n",
                param.sched_priority,
                prior);
            break;
        default :
            printf("???\n");
        break;
    }

    return EXIT_SUCCESS;
}
```

Pour le moment nous n'avons pas encore vu comment modifier notre ordonnancement, aussi le processus est-il toujours dans le mode `OTHER`. Toutefois, nous pouvons lancer un sous-shell avec la commande `nice` pour modifier la priorité dynamique (*via* la valeur de gentillesse).

```
$ ./exemple_ordonnancement 0
OTHER : Priorité statique = 0 dynamique = 0
$ nice sh
$ ./exemple_ordonnancement 0
OTHER : Priorité statique = 0 dynamique = 10
$ exit
$
```

Rappelons que la valeur affichée en tant que priorité dynamique est en fait la valeur de gentillesse, et qu'elle est donc plus élevée moins le processus est prioritaire (contrairement aux priorités statiques des processus temps-réel).

Nous allons finalement nous intéresser à l'appel-système permettant de modifier l'ordonnancement d'un processus, **`sched_setscheduler()`**. Celui-ci est déclaré ainsi :

```
int sched_setscheduler (pid_t pid,
                        int ordonnancement,
                        const struct sched_param param);
```

Nous pouvons modifier l'ordonnancement associé à un processus en cours d'exécution. La structure `sched_param` sert à préciser la priorité dynamique. Cet appel-système nécessite la capacité `CAP_SYS_NICE`. Aussi, le programme suivant doit-il être installé Set-UID *root*. Nous revenons ensuite sous l'identité effective de l'utilisateur ayant lancé le programme avant d'exécuter la commande qu'il fournit en argument. Nous passons toujours en ordonnancement RR, car le mode FIFO n'aurait pas d'intérêt particulier pour nos expériences.

exemple_setscheduler.c :

```
    #include <sched.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    void
syntaxe (char * nom)
{
    fprintf(stderr, "Syntaxe %s priorité commande...\n", nom);
    exit(EXIT_FAILURE);
}

    int
main (int argc, char * argv[])
{
    int                prior;
    struct sched_param param;

    if ((argc < 3) || (sscanf(argv[1], "%d", & prior) != 1))
        syntaxe(argv[0]);
    param.sched_priority = prior;
    if (sched_setscheduler(0, SCHED_RR, & param) < 0) {
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }
    if (seteuid(getuid()) < 0) {
        perror("seteuid");
        exit(EXIT_FAILURE);
    }
    execvp(argv[2], argv + 2);
    perror("execvp");
    return EXIT_FAILURE;
}
```

Dans notre premier exemple, nous allons simplement lancer un shell en ordonnancement temps-réel.

```
$ su
Password:
# chown root.root exemple_setscheduler
# chmod u+s exemple_setscheduler
# ls -l exemple_setscheduler
-rwsrwxr-x   1 root   root    24342 Dec  9 19:40 exemple_setscheduler
# exit
$ ./exemple_setscheduler 10 sh
$ ./exemple_ordonnancement 0
RR : Priorité = 10, intervalle = 0.000150000 s.
$
```

Bon, jusque-là, tout va bien, le shell s'exécute avec la priorité voulue et sous l'ordonnancement RR. Nous pouvons aussi le vérifier avec le programme `exemple_getscheduler` écrit plus haut.

Nous allons maintenant passer à un exercice plus acrobatique. Nous allons faire boucler un processus jusqu'à ce qu'un signal engendré par l'appel-système `alarm()` le tue. Ce sera représentatif du risque que court un développeur pendant la mise au point d'une application utilisant des processus temps-réel. Le bouclage ici sera limité à 5 secondes, mais on pourrait se trouver dans une situation où tout le temps CPU est phagocyté par un processus incontrôlé.

exemple_boucle.c :

```
    #include <unistd.h>
    int
main (void)
{
    alarm(5);
    while (1)
        ;
    return EXIT_SUCCESS
}
```

Le compte-rendu d'exécution ci-dessous ne présente pas un intérêt particulier. Par contre, il est très utile d'effectuer soi-même la manipulation.

```
$ ./exemple_setscheduler 10 sh
$ ./exemple_ordonnancement
RR : Priorité = 10, intervalle = 0.000150000 s.
$ ./exemple_boucle
Alarm clock
$ exit
exit
$
```

Lorsqu'on invoque ce programme depuis un terminal situé sur le serveur X fonctionnant sur la même machine (comme c'est le cas sur la majeure partie des stations Linux), le système est complètement gelé pendant 5 secondes. Impossible de revenir sur un écran de texte avec Contrôle-Alt-F1, et même le pointeur de la souris reste immobile !

Nous essayons alors de lancer un autre Xterm avec une priorité supérieure, et de faire exécuter la commande `top -d1` pour voir l'état des processus toutes les secondes. Dès qu'on relance `exemple_boucle`, tout se fige à nouveau pendant 5 secondes, y compris l'affichage du terminal Xterm placé à une priorité supérieure ! C'est normal, car c'est le serveur X11 qui doit faire le rafraîchissement d'écran, et il tourne sous un ordonnancement OTHER.

L'expérience suivante consiste alors à se connecter à distance depuis le réseau avec un `telnet`. Nous relançons un shell de priorité supérieure et nous exécutons `top -d1`. Le lancement de `exemple_boucle` sur la machine Linux gèle à nouveau la mise à jour des données sur le shell distant ! En cherchant un peu, nous nous rendons compte que le démon `telnetd` qui gère la connexion distante sur la machine s'exécute encore sous un ordonnancement OTHER. Même si le processus `top` continue de fonctionner effectivement pendant l'exécution de `exemple_boucle`, les informations ne sont pas transmises au terminal distant pendant cette période.

Finalement, nous basculons sur des terminaux « texte » virtuels (avec Contrôle-Alt-F1), et nous nous connectons sur les deux premiers terminaux . Le basculement s'effectue avec Alt-F1 et Alt-F2. Lorsqu'on place un shell à la priorité 10 sur l'un et 20 sur l'autre, et quand on exécute `top -d1` sur le second, on peut finalement lancer `exemple_boucle` sur un terminal, basculer vers l'autre, et observer sa progression avec `top` qui continue de fonctionner. Comme c'est le noyau qui assure la commutation entre les terminaux virtuels et le dialogue avec eux, il n'y a plus de problèmes d'ordonnancement. Nous avons enfin trouvé un moyen de conserver la main, même avec un processus temps-réel qui boucle.

Si on désire travailler sous X-Window, on peut utiliser un serveur X11 fonctionnant sur une autre machine. Sinon, il faut passer le serveur en ordonnancement temps-réel. Pour cela, on invoque :

```
$ ./exemple_setscheduler 10 startx
```

On ouvre un Xterm et on diminue la priorité du shell :

```
$ ./exemple_setscheduler 5 sh
$
```

Le nouveau shell peut alors servir à lancer l'application à déboguer ; on pourra toujours la tuer depuis un autre Xterm fonctionnant avec la priorité 10.

Pour finir, précisons le prototype de l'appel-système **`sched_yield()`** dont nous avons déjà parlé, et qui permet à un processus de libérer volontairement le processeur.

```
int sched_yield(void);
```

Rappelons que le processus appelant ne sera remplacé que par un autre, Prêt, de même priorité. L'ordonnanceur n'élira jamais un processus de priorité inférieure. Quant à un éventuel processus de priorité supérieure, il prendra automatiquement le contrôle dès qu'il sera prêt, sans que `sched_yield()` n'ait besoin d'être invoqué. Cet appel-système est en réalité rarement utile. Dans l'ordonnancement RR, il n'a pas lieu d'être car la commutation se fait automatiquement au bout d'un certain délai ou si le processus s'endort. Dans l'ordonnancement FIFO, deux processus de même priorité ont un comportement plus difficile à prévoir, puisque la commutation intervient uniquement lorsque le processus en cours d'exécution s'endort sur un appel-système bloquant.

Lors d'un `fork()`, Posix.1b mentionne que le processus père doit continuer à s'exécuter avant son fils. Si l'ordonnancement est FIFO, le père peut même s'exécuter jusqu'à la fin sans que

son fils n'ait eu le temps de démarrer. Cela pose un sérieux problème si le fils doit basculer sur une priorité supérieure à celle de son père. On pourrait imaginer utiliser `sched_yield()` immédiatement après `fork()` pour donner la main au processus fils afin qu'il invoque `sched_setscheduler()`. Toutefois, ce fonctionnement est aléatoire car nous ne sommes pas sûr que le processus fils soit prêt immédiatement (si le noyau doit envoyer des pages sur le périphérique de swap pour créer le contexte du nouveau processus par exemple), et le processeur peut être réattribué immédiatement au père.

Il vaut mieux dans ce cas s'en tenir à ce qui est défini par Posix.1b, c'est-à-dire placer la modification de priorité dans le code du processus père. On n'utilisera pas

```
if ((pid = fork()) < 0)
    erreur_fatale();
if (pid == 0) {
    /* fils */
    sched_setscheduler(0, SCHED_FIFO, &mes_parametres) ;
    /* ... */
} else {
    sched_yield();
    /* ... */
}
```

mais plutôt

```
if ((pid = fork()) < 0)
    erreur_fatale();
if (pid == 0) {
    /* Fils */
    /* ... */
} else {
    sched_setscheduler(pid, SCHED_FIFO, & parametres_fils) ;
    /* ... */
}
```

Notons que le comportement de `sched_setscheduler()` est parfaitement défini par SUSv3 lorsqu'on modifie la priorité d'un autre processus. Si ce dernier devient plus prioritaire que l'appelant, il est exécuté immédiatement, avant le retour de l'appel-système.

Conclusion

L'ordonnancement des processus est un domaine très intéressant et plus facile à appréhender qu'on pourrait le croire au premier abord. La présentation générale des principes d'un ordonnanceur est décrite dans [TANENBAUM 1997] *Operating Systems, Design and Implementation*.

La description la plus complète des ordonnancements temps-réel est probablement celle qu'on trouve dans [GALLMEISTER 1995] *Posix.4 Programming For The Real World*.

12

Programmation multithread

Les threads représentent un concept relativement nouveau dans le domaine de la programmation. Il s'agit d'une manière différente d'aborder la conception multitâche. Linux implémente les mécanismes qui, mis en œuvre par le noyau et des fonctions de bibliothèque, permettent d'accéder à la puissance des threads avec la portabilité de la norme SUSv3.

Ces fonctionnalités étant standardisées, elles ont donné naissance au terme *Pthread* pour représenter les threads compatibles avec la norme Posix.1c qui les décrivait en premier lieu.

Ce chapitre présentera les notions et les routines essentielles de la programmation multithread. Naturellement, nous ne pourrons pas inspecter en profondeur toutes les implications de ce mode de travail. Pour plus de détails, notamment en ce qui concerne les conditions de synchronisation permettant d'éviter les blocages, on se reportera par exemple à [BUTENHOF 1997] *Programming with POSIX Threads*

Présentation

Le mot *thread* peut se traduire par « fil d'exécution », c'est-à-dire un déroulement particulier du code du programme qui se produit parallèlement à d'autres entités en cours de progression. Les threads sont généralement présentés en premier lieu comme des processus allégés ne réclamant que peu de ressources pour les changements de contexte. Il faut ajouter à ceci un point important : les différents threads d'une application partagent un même espace d'adressage en ce qui concerne leurs données. La vision du programmeur est d'ailleurs plus orientée sur ce dernier point que sur la simplicité de commutation des tâches.

En première analyse, on peut imaginer les threads comme des processus partageant les mêmes données statiques et dynamiques. Chaque thread dispose personnellement d'une pile et d'un contexte d'exécution contenant les registres du processeur et un compteur d'instruction. Les méthodes de communication entre les threads sont alors naturellement plus simples que les communications entre processus. En contrepartie, l'accès concurrentiel aux mêmes données

nécessite une synchronisation pour éviter les interférences, ce qui complique certaines portions de code.

Les threads définis par la norme SUSv3 sont indépendants de l'implémentation sous-jacente dans le système d'exploitation. Les applications sont donc portables sur des systèmes n'implémentant pas la notion de processus ni leurs méthodes de communication.

Les threads ne sont intéressants que dans les applications assurant plusieurs tâches en parallèle. Si chacune des opérations effectuées par un logiciel doit attendre la fin d'une opération précédente avant de pouvoir démarrer, il est totalement inutile d'essayer une approche multi-thread.

Implémentation

Pour implémenter les fonctionnalités de la norme SUSv3, il existe essentiellement deux possibilités : l'implémentation dans l'espace du noyau, et celle dans l'espace de l'utilisateur. En implémentation noyau, chaque thread est vu individuellement par le noyau ; il s'agit approximativement d'un processus indépendant partageant son espace d'adressage avec les autres threads de la même application. En implémentation utilisateur, l'application n'est constituée que d'un seul processus, et la répartition en différents threads est assurée par une bibliothèque indépendante du noyau. Chaque implémentation a ses avantages et ses défauts :

Point de vue	Implémentation dans l'espace du noyau	Implémentation dans l'espace de l'utilisateur
Implémentation des fonctionnalités SUSv3.	Nécessite la présence d'appels-système spécifiques, qui n'existent pas nécessairement sur toutes les versions du noyau.	Portable de système Unix en système Unix sans modification du noyau.
Création d'un thread.	Nécessite un appel-système.	Ne nécessite pas d'appel-système, est donc moins coûteuse en ressource que l'implémentation dans le noyau.
Commutation entre deux threads.	Commutation par le noyau avec changement de contexte.	Commutation assurée dans la bibliothèque sans changement de contexte, est donc plus légère.
Ordonnancement des threads.	Chaque thread dispose des mêmes ressources CPU que les autres processus du système.	Utilisation globale des ressources CPU limitée à celle du processus d'accueil.
Priorités des tâches.	Chaque thread peut s'exécuter avec une priorité indépendante des autres, éventuellement en ordonnancement temps-réel.	Les threads ne peuvent s'exécuter qu'avec des priorités inférieures à celle du processus principal.
Parallélisme.	Sur une machine SMP, le noyau peut répartir les threads sur différents processeurs et profiter ainsi du véritable parallélisme.	Les threads sont condamnés à s'exécuter sur un seul processeur puisqu'ils sont contenus dans un unique processus.

Sous Linux, l'implémentation usuelle est effectuée dans l'espace noyau à l'aide de l'appel-système `__clone()`. Celui-ci est déclaré dans `<sched.h>` ainsi :

```
int __clone (int (* fonction) (void * arg),
             void * pile, int attributs, void * arg);
```

Cet appel-système est un genre de `fork()` grâce auquel le processus fils partage l'espace d'adressage de son père. Toutefois cette fonction ne concerne en principe jamais le programmeur applicatif, car une bibliothèque fournit les fonctionnalités SUSv3 que nous allons décrire dans le reste de ce chapitre.

Jusqu'au noyau 2.4, la principale bibliothèque implémentant les threads s'appelait LinuxThreads. Puissante, elle souffrait toutefois de quelques limitations, notamment en ce qui concerne la gestion des signaux. Depuis le noyau 2.6 une nouvelle bibliothèque a été intégrée à la bibliothèque C : la NPTL (*Native Posix Thread Library*). En outre, le noyau offre un support nettement amélioré en ce qui concerne la création et l'ordonnancement des threads.

Il existe d'autres implémentations des Pthreads sous Linux, notamment *PCthreads* qui fonctionne dans l'espace utilisateur, mais elles sont moins répandues que NPTL.

Pour employer les fonctionnalités de la bibliothèque NPTL, il faut inclure l'en-tête `<pthread.h>` dans les fichiers source, ajouter l'option `–pthread` sur la ligne de commande du compilateur gcc. Nous ajoutons ces options dans le fichier `Makefile` pour l'utiliser systématiquement avec tous les programmes de ce chapitre.

> L'option `–pthread` a deux effets principaux : lors de la compilation elle permet de choisir une implémentation fonctionnant en contexte multithread pour toutes les routines de la bibliothèque C que l'on invoque (c'était le rôle de la constante symbolique `_REENTRANT` avant la NPTL). Pendant l'édition des liens, cette option s'assure que l'on utilise la bibliothèque implémentant les fonctions de gestion des threads (ce que réalisait l'option `–lpthread` auparavant).

Création de threads

Il existe *grosso modo* des équivalents aux appels-système `fork()`, `wait()` et `exit()`, qui permettent dans un contexte multithread de créer un nouveau thread, d'attendre la fin de son exécution, et de mettre fin au thread en cours.

Un type opaque `pthread_t` est utilisé pour distinguer les différents threads d'une application, à la manière des PID qui permettent d'identifier les processus. Dans la bibliothèque NPTL, le type `pthread_t` est implémenté sous forme d'un `unsigned long`, mais sur d'autres systèmes il peut s'agir d'une structure. On se disciplinera donc pour employer systématiquement la fonction **`pthread_equal()`** lorsqu'on voudra comparer deux identifiants de threads.

```
int pthread_equal (pthread_t thread_1, pthread_t thread_2);
```

Cette fonction renvoie une valeur non nulle s'ils sont égaux.

Lors de la création d'un nouveau thread, on emploie la fonction **`pthread_create()`**. Celle-ci donne naissance à un nouveau fil d'exécution, qui va démarrer en invoquant la routine dont on passe le nom en argument. Lorsque cette routine se termine, le thread est éliminé. Cette routine fonctionne donc un peu comme la fonction `main()` des programmes C. Pour cette raison, le fil d'exécution original du processus est nommé thread principal (*main thread*). Le prototype de `pthread_create()` est le suivant :

```
int pthread_create (pthread_t * thread, pthread_attr_t * attributs,
                    void * (* fonction) (void * argument),
                    void * argument);
```

Le premier argument est un pointeur qui sera initialisé par la routine avec l'identifiant du nouveau thread. Le second argument correspond aux attributs dont on désire doter le nouveau thread. Ces attributs seront détaillés dans la prochaine section. En général, on transmettra en second argument un pointeur `NULL`, car le thread reçoit alors les attributs standard par défaut.

Le troisième argument est un pointeur représentant la fonction principale du nouveau thread. Celle-ci est invoquée dès la création du thread et reçoit en argument le pointeur passé en dernière position dans `pthread_create()`. Le type de l'argument étant `void *`, on pourra le transformer en n'importe quel type de pointeur pour passer un argument au thread. On pourra même employer une conversion explicite en `int` pour transmettre une valeur. Cet argument est généralement un numéro permettant au thread de déterminer le travail qu'il doit accomplir – en employant la même fonction principale pour plusieurs threads –, mais cela peut aussi être un pointeur sur une structure que le nouveau thread doit manipuler avant de se terminer.

Cette routine renvoie zéro si elle réussit et une valeur non nulle sinon, correspondant à l'erreur survenue. En effet, comme l'essentiel des fonctions de la bibliothèque NPTL, `pthread_create()` ne remplit pas nécessairement la variable globale `errno`[1].

Lorsque la fonction principale d'un thread se termine, celui-ci est éliminé. Cette fonction doit renvoyer une valeur de type `void *` qui pourra être récupérée dans un autre fil d'exécution. Il est aussi possible d'invoquer la fonction **`pthread_exit()`**, qui met fin au thread appelant tout en renvoyant le pointeur `void *` passé en argument. On ne doit naturellement pas invoquer `exit()`, qui mettrait fin à toute l'application et pas uniquement au thread appelant.

```
void pthread_exit (void * retour);
```

Lorsque cette routine est appelée, toutes les fonctions de nettoyage final que nous observerons ultérieurement sont invoquées dans l'ordre inverse de leur enregistrement.

Pour récupérer la valeur de retour d'un thread terminé, on utilise la fonction **`pthread_join()`**. Celle-ci suspend l'exécution du thread appelant jusqu'à la terminaison du thread indiqué en argument. Elle remplit alors le pointeur passé en seconde position avec la valeur de retour du thread fini.

```
int pthread_join (pthread_t thread, void ** retour);
```

Lorsqu'on désire employer une valeur de type entier comme code de retour, il est important de procéder en deux étapes pour la récupérer depuis le type `void *`. Ceci permet de conserver la portabilité du programme, même si la taille d'un `void *` est plus grande que celle d'un `int`. Nous rencontrerons cette situation dans les programmes d'exemples décrits ci-dessous.

Dans certaines conditions, un thread peut être annulé par un autre thread, un peu à la manière d'un processus qui peut être tué par un signal. Dans ce cas, le thread terminé n'a pu renvoyer de valeur, aussi la variable `* retour` prend-elle une valeur particulière : **`PTHREAD_CANCEL`**. Cette constante ne peut pas correspondre à une adresse valide, pas plus qu'à la valeur `NULL` d'ailleurs.

La fonction `pthread_join()` peut échouer si le thread attendu n'existe pas, s'il est détaché, comme nous le verrons plus bas, ou si un risque de blocage se présente – par exemple si un thread demande à attendre sa propre fin.

1. Les relations entre la bibliothèque Pthreads et la variable `errno` seront traitées plus loin en détail.

Nous pouvons déjà employer ces fonctions pour écrire un petit programme simpliste avec plusieurs fils d'exécution incrémentant un compteur jusqu'à ce qu'il atteigne 40, puis qui se termine.

exemple_create.c :

```
    #include <pthread.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>

    #define NB_THREADS   5

    void * fn_thread (void * numero);

    static int compteur = 0;

    int
main (void)
{
    pthread_t thread[NB_THREADS];
    int       i;
    int       ret;

    for (i = 0; i < NB_THREADS; i ++)
        if ((ret = pthread_create(& thread[i],
                                  NULL,
                                  fn_thread,
                                  (void *) i)) != 0) {
            fprintf(stderr, "%s", strerror(ret));
            exit(EXIT_FAILURE);
        }
    while (compteur < 40) {
        fprintf(stdout, "main : compteur = %d \n", compteur);
        sleep(1);
    }
    for (i = 0; i < NB_THREADS; i ++)
        pthread_join(thread[i], NULL);
    return EXIT_SUCCESS;
}

    void *
fn_thread (void * num)
{
    int numero = (int) num;
    while (compteur < 40) {
        usleep(numero * 100000);
        compteur ++;
        fprintf(stdout, "Thread %d : compteur = %d \n",
                                 numero, compteur);
    }
    pthread_exit(NULL);
}
```

Lors de son exécution, ce programme montre bien l'enchevêtrement des différents threads :

```
$ ./exemple_create
main : compteur = 0
Thread 0 : compteur = 1
Thread 0 : compteur = 2
Thread 0 : compteur = 3
Thread 0 : compteur = 4
Thread 0 : compteur = 5
Thread 0 : compteur = 6
Thread 0 : compteur = 7
Thread 0 : compteur = 8
Thread 0 : compteur = 9
Thread 0 : compteur = 10
Thread 0 : compteur = 11
Thread 1 : compteur = 12
Thread 0 : compteur = 13
Thread 0 : compteur = 14
Thread 0 : compteur = 15
Thread 0 : compteur = 16
Thread 0 : compteur = 17
Thread 0 : compteur = 18
Thread 0 : compteur = 19
Thread 0 : compteur = 20
Thread 0 : compteur = 21
Thread 0 : compteur = 22
Thread 2 : compteur = 23
Thread 0 : compteur = 24
Thread 1 : compteur = 25
Thread 0 : compteur = 26
Thread 0 : compteur = 27
Thread 0 : compteur = 28
Thread 0 : compteur = 29
Thread 0 : compteur = 30
Thread 0 : compteur = 31
Thread 0 : compteur = 32
Thread 0 : compteur = 33
Thread 0 : compteur = 34
Thread 3 : compteur = 35
Thread 0 : compteur = 36
Thread 0 : compteur = 37
Thread 1 : compteur = 38
Thread 0 : compteur = 39
Thread 0 : compteur = 40
Thread 4 : compteur = 41
Thread 2 : compteur = 42
Thread 1 : compteur = 43
Thread 3 : compteur = 44
$
```

Cette application est très mal conçue, car les différents threads modifient la même variable globale sans se préoccuper les uns des autres. Et c'est justement l'essence même de la programmation multithread d'éviter ce genre de situation, comme nous le verrons dans les prochains paragraphes.

Le programme suivant n'utilise qu'un seul thread autre que le fil d'exécution principal ; il s'agit simplement de vérifier le comportement des fonctions `pthread_join()` et `pthread_exit()`. Nous sous-traitons la lecture d'une valeur au clavier dans un fil d'exécution secondaire. Le fil principal pourrait en profiter pour réaliser d'autres opérations.

exemple_join.c :

```
    #include <pthread.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>

    void *
fn_thread (void * inutile)
{
    char chaine[128];
    int  i = 0;
    fprintf(stdout, "Thread : entrez un nombre :");
    while (fgets(chaine, 128, stdin) != NULL)
        if (sscanf(chaine, "%d", & i) != 1)
            fprintf(stdout, "un nombre SVP :");
        else
            break;
    pthread_exit((void *) i);
}

    int
main (void)
{
    int       i;
    int       ret;
    void *    retour;
    pthread_t thread;

    if ((ret = pthread_create(& thread, NULL, fn_thread, NULL)) != 0) {
        fprintf(stderr, "%s\n", strerror(ret));
        exit(EXIT_FAILURE);
    }
    pthread_join(thread , & retour);
    if (retour != PTHREAD_CANCELED) {
        i = (int) retour;
        fprintf(stdout, "main : valeur lue = %d\n", i);
    }
    return(EXIT_SUCCESS);
}
```

Sans surprise, l'exécution se déroule ainsi :

```
$ ./exemple_join
Thread : entrez un nombre :4767
main : valeur lue = 4767
$
```

On notera la méthode correcte employée pour récupérer la valeur de retour dans l'appel `pthread_join()`, en utilisant d'abord un pointeur `void *` temporaire, qu'on transforme ensuite en `int`.

Lorsqu'un thread ne renvoie pas de valeur intéressante et qu'il n'a pas besoin d'être attendu par un autre thread, on peut employer la fonction **`pthread_detach()`**, qui lui permet de disparaître automatiquement du système quand il se termine. Cela autorise la libération immédiate des ressources internes à la bibliothèque concernant ce thread (pile et variables automatiques). Le mécanisme n'est pas sans rappeler le passage des processus à l'état Zombie en attendant qu'on lise leur code de retour et l'emploi de `SIG_IGN` pour le signal `SIGCHLD` du processus parent.

```
int pthread_detach (pthread_t thread);
```

Un thread peut très bien invoquer `pthread_detach()` à propos d'un autre thread de l'application. Contrairement aux processus, il n'y a pas de notion de hiérarchie chez les threads ni d'autorisations particulières pour modifier les paramètres d'un autre fil d'exécution. Cette fonction échoue si le thread n'existe pas ou s'il est déjà détaché.

Il n'est pas possible d'attendre, avec `pthread_join()`, la fin d'un thread donné parmi tous ceux qui se déroulent. Ce genre de fonctionnalité pourrait être utile pour attendre que tous les threads d'un certain ensemble se terminent. On peut obtenir le même résultat en utilisant des threads détachés qui décrémentent un compteur global avant de se terminer, le thread principal surveillant alors l'état de ce compteur. Des précautions devront être prises pour l'accès au compteur global, comme nous le verrons plus tard.

Pour connaître son propre identifiant, un thread invoque la fonction **`pthread_self()`**, qui lui renvoie une valeur de type `pthread_t` :

```
pthread_t pthread_self (void);
```

Il lui est alors possible de comparer, avec `pthead_equal()`, son identité avec une variable globale indiquant une tâche à accomplir. Ainsi un programme peut utiliser ce genre de principe :

```
    static pthread_t    thr_dialogue_radar;
    static pthread_t    thr_dialogue_automate;
    static pthread_t    thr_dialogue_radar_2;
    int
main (void)
{
    pthread_create (& thr_dialogue_radar,    NULL, dialogue, NULL);
    pthread_create (& thr_dialogue_automate, NULL, dialogue, NULL);
    pthread_create (& thr_dialogue_radar_2,  NULL, dialogue, NULL);
  [...]
}

    void *
dialogue (void * inutile)
{
    initialisation_generale_dialogue();
    if (pthread_equal(pthread_self(), thr_dialogue_radar))
        initialisation_specifique_radar();
    else if (pthread_equal(pthread_self(), thr_dialogue_automate))
        initialisation_specifique_automate();
```

```
    else if (pthread_equal(pthread_self(), thr_dialogue_radar_2))
        initialisation_specifique_radar_2();
    [...]
}
```

Dans ce cas précis, on aurait avantageusement obtenu le même résultat en utilisant l'argument de la fonction `dialogue()`. Toutefois, l'intérêt de la comparaison avec une variable globale `pthread_t` est de permettre d'adapter le comportement au sein de sous-routines profondément imbriquées, jusqu'auxquelles on ne désire pas faire descendre l'argument de `dialogue()`.

Rappelons-nous que le type `pthread_t` est opaque et ne permet pas d'affectation directe. Il n'est donc pas possible de mémoriser l'identifiant du thread principal, puisqu'il n'est pas créé par `pthread_create()`. Si on en a besoin, il faut procéder en créant quand même un thread nouveau :

```
    pthread_t  thr_principal;

    int
main (void)
{
    void * retour;
    pthread_create(& thr_principal, NULL, suite_application, NULL);
    pthread_join(thr_principal, & retour);
    return((int) retour);
}

    void *
suite_application (void * inutile)
{
  [...]
   pthread_exit((void *) 0);
}
```

Attributs des threads

Chaque thread est doté d'un certain nombre d'attributs, regroupés dans un type opaque `phtread_attr_t`. Les attributs sont fixés lors de la création du thread grâce au second argument de `pthread_create()`. Lorsque les attributs par défaut sont suffisants, on passe généralement un pointeur `NULL` sur cet argument. Sinon, il faut passer un pointeur sur un objet `pthread_attr_t` qui aura été préalablement configuré correctement.

Dans un premier temps, il faut invoquer la fonction **`pthread_attr_init()`** en lui transmettant un pointeur sur une variable de type `pthread_attr_t`. Cette fonction peut servir – dans d'autres implémentations que celle de Linux – à allouer des zones de mémoire au sein du type opaque `pthread_attr_t`, aussi est-il indispensable de l'invoquer avant d'utiliser les attributs.

```
int pthread_attr_init (pthread_attr_t * attributs);
```

Une fois les attributs initialisés, on utilise les fonctions `pthread_attr_getXXX()` et `pthread_attr_setXXX()` pour consulter ou changer l'attribut correspondant. Les attributs vont être décrits ci-dessous. L'objet de type `pthread_attr_t` est alors prêt à être utilisé dans `pthread_create()`. Le thread créé est configuré en fonction des attributs indiqués, et l'objet `pthread_attr_t` peut être à nouveau modifié pour préparer la création d'un autre thread. Une fois qu'on n'en a plus

besoin, cet objet peut être détruit en employant la fonction **pthread_attr_destroy()**, qui peut libérer des données dynamiques internes. L'invocation de cette fonction est donc indispensable pour assurer la portabilité du programme, même si elle n'a pas d'utilité dans l'implémentation NPTL actuelle. Les threads créés précédemment ne sont pas concernés par la destruction de l'objet attribut, ce dernier a été utilisé une fois pour toutes au moment de la création du thread, mais n'a plus de liaison avec lui.

```
int pthread_attr_destroy (pthread_attr_t * attributs);
```

Les fonctions permettant donc de modifier un attribut ou de lire sa valeur se nomment **pthread_attr_setXXX()** et **pthread_attr_getXXX()**, le `XXX` indiquant l'attribut concerné. Ces fonctions prennent un pointeur sur un objet `pthread_attr_t` en premier argument. Le second argument est une valeur pour les fonctions `pthread_attr_setXXX()`, et un pointeur sur une valeur pour les routines `pthread_attr_getXXX()`. La valeur de l'attribut est généralement un entier, mais il existe quelques exceptions. Nous allons faire une présentation rapide des fonctions de configuration des attributs. Si la valeur indiquée pour l'attribut n'est pas acceptable, la fonction échoue avec l'erreur `EINVAL`.

Rappelons que ces fonctions permettent uniquement de configurer un ensemble d'attributs en vue de les transmettre à `pthread_create()`. Il n'est pas possible d'employer ces routines pour modifier les attributs concernant un thread déjà créé.

L'attribut **detachstate** correspond au détachement du thread avant son démarrage. Nous avons déjà vu que `pthread_detach()` permet de le modifier de manière dynamique.

```
int pthread_attr_getdetachstate (const pthread_attr_t * attributs,
                                 int * valeur);
int pthread_attr_setdetachstate (pthread_attr_t * attributs,
                                 int valeur);
```

La valeur de cet attribut, disponible dans toutes les implémentations Posix.1c, peut consister en l'une des constantes symboliques suivantes :

Nom	Signification
`PTHREAD_CREATE_JOINABLE`	Configuration par défaut, la valeur de retour du thread sera conservée jusqu'à ce qu'elle soit consultée par un autre thread.
`PTHREAD_CREATE_DETACHED`	Lors de la terminaison du thread, toutes ses ressources sont libérées immédiatement, il n'y a pas de valeur de retour valide.

Les attributs **stackaddr** et **stacksize** permettent de configurer la pile utilisée par un thread. Comme nous l'avons déjà indiqué, chaque thread dispose d'une pile personnelle, dans laquelle sont allouées toutes ses variables automatiques. Il peut être parfois nécessaire de réclamer une pile de dimension plus grande que celle qui est fournie par défaut si le thread à créer fait un large usage de fonctions récursives, par exemple.

À l'inverse, il peut être nécessaire de réduire la taille de la pile accordée à un thread. Par défaut, la NPTL emploie des piles de 8 Mo, ce qui est bien souvent très supérieur aux besoins des threads, et ce qui limite le nombre de threads utilisables (de l'ordre de 256 threads). En réduisant la taille de la pile à 16 Ko, on peut employer plusieurs milliers, voire dizaines de milliers de threads.

Comme nous le verrons dans le prochain chapitre, la pile d'un processus habituel n'est contrainte que par la limite de la zone nommée *tas*, dans laquelle les variables dynamiques sont allouées. En principe, la pile d'un tel processus pourrait croître jusqu'à remplir l'essentiel de l'espace d'adressage du programme, soit environ 3 Go. Dans le cas d'un programme multithread, les différentes piles doivent être positionnées à des emplacements figés dès la création des threads, ce qui impose *ipso facto* des limites de taille puisqu'elles ne doivent pas se rejoindre.

Les routines citées ci-dessous permettent respectivement de lire ou d'indiquer l'adresse de départ et la taille maximale de la pile du thread. Elles sont disponibles dans la bibliothèque Pthreads si les constantes symboliques **_POSIX_THREAD_ATTR_STACKADDR** et **_POSIX_THREAD_ATTR_STACKSIZE** sont définies dans `<unistd.h>`. La dimension minimale de la pile est disponible dans la constante symbolique **PTHEAD_STACK_MIN** (16 Ko sur PC sous Linux).

Naturellement, l'emploi de ces attributs est assez pointu, et on le réservera aux applications pour lesquelles ils sont réellement indispensables.

```
int pthread_attr_getstackaddr (const pthread_attr_t * attributs,
                               void ** valeur);
int pthread_attr_setstackaddr (pthread_attr_t * attributs,
                               void * valeur);
int pthread_attr_getstacksize (const pthread_attr_t * attributs,
                               size_t * valeur);
int pthread_attr_setstacksize (pthread_attr_t * attributs,
                               size_t valeur);
```

Les attributs `schedpolicy`, `schedparam`, `scope` et `inheritsched` concernent l'ordonnancement des threads. Ils sont disponibles si la constante symbolique **_POSIX_THREAD_PRIORITY_SCHEDULING** est définie avec une valeur vraie dans `<unistd.h>`.

```
int pthread_attr_getschedpolicy (const pthread_attr_t * attributs,
                                 int * valeur);
int pthread_attr_setschedpolicy (pthread_attr_t * attributs, int valeur);
```

L'attribut **schedpolicy** correspond à la méthode d'ordonnancement employée pour le thread. Les valeurs possibles sont semblables à celles qui ont déjà été étudiées au chapitre précédent, avec l'appel-système `sched_setscheduler()` :

Nom	Signification
SCHED_OTHER	Ordonnancement classique
SCHED_RR	Séquencement temps-réel avec l'algorithme Round Robin
SCHED_FIFO	Ordonnancement temps-réel FiFO

On peut aussi employer la fonction `pthread_setschedparam()` que nous verrons ci-dessous pour modifier la politique de séquencement après création du thread. Les ordonnancements temps-réel nécessitent un UID effectif nul, sinon la fonction `pthread_create()` échouera avec l'erreur `EPERM`.

L'attribut **schedparam** contient principalement la priorité du processus, telle que nous l'avons vue dans le chapitre précédent.

```
int pthread_attr_getschedparam (const pthread_attr_t * attributs,
                                struct sched_param * params);
int pthread_attr_setschedparam (pthread_attr_t * attributs,
                                const struct sched_param * params);
```

L'attribut schedparam, qui ne concerne que les ordonnancements temps-réel (RR et FIFO), ainsi que l'attribut schedpolicy peuvent être consultés ou modifiés durant l'exécution du thread à l'aide des fonctions **pthread_setschedparam()** et **pthread_getschedparam()** :

```
int pthread_setschedparam (pthread_t thread,
                           int ordonnancement,
                           const struct sched_param * params);
int pthread_getschedparam (pthread_t thread,
                           int * ordonnancement,
                           struct sched_param * params);
```

L'attribut **scope** n'est pas vraiment configurable avec la bibliothèque NPTL. Il sert dans les implémentations hybrides reposant en partie sur un ordonnancement par le noyau – comme c'est le cas sous Linux – et en partie sur une bibliothèque dans l'espace utilisateur. Dans ce cas, cet attribut peut prendre l'une des valeurs suivantes :

Nom	Signification
PTHREAD_SCOPE_SYSTEM	Cette valeur réclame un ordonnancement du thread en concurrence avec tous les processus du système. Le séquenceur utilisé est donc celui du noyau.
PTHREAD_SCOPE_PROCESS	Non supporté avec NPTL, cet ordonnancement oppose les threads les uns aux autres, au sein du même processus.

L'attribut scope est donc relatif aux priorités des threads : dans un cas vis-à-vis du système, et dans l'autre cas, interne au processus qui les accueille.

```
int pthread_attr_getscope (const pthread_attr_t * attributs,
                           int * valeur);
int pthread_attr_setscope (pthread_attr_t * attributs, int valeur);
```

Sous Linux, l'invocation

```
pthread_attr_setscope (& attributs, PTHREAD_SCOPE_PROCESS);
```

échoue avec l'erreur ENOTSUP.

Finalement, l'attribut **inheritsched** signale si le thread dispose de sa propre configuration d'ordonnancement, comme c'est le cas par défaut, ou si les attributs schedparam et schedpolicy sont ignorés, au profit de l'ordonnancement du thread qui l'a créé. Les valeurs de cet attribut peuvent être :

Nom	Signification
PTHREAD_EXPLICIT_SCHED	L'ordonnancement est spécifique au thread créé (par défaut).
PTHREAD_INHERIT_SCHED	L'ordonnancement est hérité du thread créateur.

```
int pthread_attr_getinheritsched (const pthread_attr_t * attributs,
                                  int * valeur);
int pthread_attr_setinheritsched (pthread_attr_t * attributs, int valeur);
```

Déroulement et annulation d'un thread

Un thread peut vouloir annuler un autre thread. Il envoie alors une demande d'annulation, qui sera prise en compte ou non, en fonction de la configuration du thread récepteur. Lorsqu'un thread est annulé, il se comporte exactement comme s'il invoquait la fonction `pthread_exit()` avec l'argument spécial `PTHREAD_CANCELED`. Lorsque le thread annulé se termine, il exécute toutes les fonctions de terminaison programmées, comme nous le verrons plus bas.

Le thread récepteur peut accepter la requête, la refuser ou la repousser jusqu'à atteindre un *point d'annulation* dans son exécution. Pour envoyer une demande d'annulation, on emploie la fonction **`pthread_cancel()`**, qui renvoie 0 si elle réussit, ou l'erreur `ESRCH` si le thread visé n'existe pas ou plus.

```
int pthread_cancel (pthread_t thread);
```

La suppression d'un thread est un phénomène assez subtil. On y a recours généralement lorsqu'un thread n'a plus d'intérêt parce qu'on a obtenu par un autre moyen le résultat qu'il devait fournir, ou après un abandon demandé par l'utilisateur. Prenons l'exemple d'une application recherchant un nombre qui vérifie certaines propriétés[1]. Elle peut, pour tirer profit du parallélisme d'une machine multiprocesseur, scinder son espace de recherche en plusieurs portions et déclencher une série de threads, chacun explorant sa portion personnelle. Aussitôt qu'un thread aura trouvé la valeur recherchée, on pourra annuler ses confrères.

Toutefois, un thread manipule souvent des variables globales en employant des techniques de verrouillage que nous étudierons dans les prochains paragraphes. S'il se trouve annulé brutalement au moment de la mise à jour d'une variable globale, il peut la laisser dans un état instable ou abandonner un verrou en position bloquée. Il faut donc pouvoir interdire temporairement les demandes d'annulation dans certaines portions du code.

La fonction **`pthread_setcancelstate()`** permet de configurer le comportement du thread appelant vis-à-vis d'une requête d'annulation.

```
int phtread_setcancelstate (int etat_annulation, int * ancien_etat);
```

Les valeurs possibles sont les suivantes :

Nom	Signification
`PTHREAD_CANCEL_ENABLE`	Le thread acceptera les requêtes d'annulation (comportement par défaut).
`PTHREAD_CANCEL_DISABLE`	Le thread ne tiendra pas compte des demandes d'annulation.

Les requêtes d'annulation ne sont pas mémorisées, contrairement aux signaux par exemple, ce qui fait d'un thread désactivant temporairement les requêtes pendant une zone de code critique ne se terminera pas lorsqu'il autorisera de nouveau les annulations, même si plusieurs demandes sont arrivées pendant ce laps de temps.

1. On peut imaginer par exemple la recherche par la force brute d'une clé de décryptage d'un message.

Il faut donc trouver un moyen plus souple pour empêcher l'annulation de se produire intempestivement, tout en acceptant les requêtes. Pour cela, on utilise tout simplement un mécanisme de synchronisation fondé sur un retardement des annulations jusqu'à atteindre des emplacements bien définis du code.

Le thread ainsi configuré ne se terminera pas dès la réception d'une annulation, mais continuera de s'exécuter jusqu'à atteindre un point d'annulation – que nous allons définir ci-dessous. Pour configurer ce comportement, on emploie la fonction **pthread_setcanceltype()** dont le prototype est :

```
int phtread_setcanceltype (int type_annulation, int * ancien_type);
```

Le type d'annulation peut correspondre à l'une des constantes suivantes :

Nom	Signification
PTHREAD_CANCEL_DEFERRED	Le thread ne se terminera qu'en atteignant un point d'annulation (comportement par défaut).
PTHREAD_CANCEL_ASYNCHRONOUS	L'annulation prendra effet dès réception de la requête.

La norme SUSv3 décrit quatre fonctions qui constituent des points d'annulation, c'est-à-dire des fonctions dans lesquelles un thread est susceptible de se terminer brutalement :

pthread_cond_wait() et pthread_cond_timedwait() que nous verrons dans un prochain paragraphe.

pthread_join() que nous avons déjà examinée.

pthread_testcancel().

Cette dernière fonction est déclarée ainsi :

```
void pthread_testcancel (void);
```

Dès qu'on l'invoque, le thread peut se terminer si une demande d'annulation est en attente. On voit donc que dans le cas PTHREAD_CANCEL_DEFERRED, un thread ne sera jamais interrompu au milieu d'un calcul ou dans une boucle de manipulation des données. On pourra donc répartir des appels pthread_testcancel() dans ce genre de code, aux endroits où on est sûr qu'une annulation ne présente pas de danger.

Il existe aussi dans SUSv3 un ensemble minimal d'appels-système qui représentent des points d'annulation sur toutes les implémentations des Pthreads (certains de ces appels sont optionnels dans SUSv3 et ne sont pas implémentés sous Linux) :

- accept(), aio_suspend(),
- clock_nanosleep(), close(), connect(), creat(),
- fcntl(), fsync(),
- getmsg(), getpmsg(),
- lockf(),
- mqreceive(), mqsend(), mq_timed_receive(), mq_timed_send(), msqrcv(), msqsnd(), msync(),
- nanosleep(),

- `open()`,
- `pause()`, `poll()`, `pread()`, `pselect()`, `putmsg()`, `putpmsg()`, `pwrite()`,
- `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`,
- `select()`, `sem_timedwait()`, `sem_wait()`, `send()`, `sendmsg()`, `sendto()`, `sigpause()`, `sigsuspend()`, `sigtimedwait()`, `sigwait()`, `sigwaitinfo()`, `sleep()`, `system()`,
- `tcdrain()`,
- `usleep()`,
- `wait()`, `waitid()`, `waitpid()`, `write()`, `writev()`.

Ces fonctions ont en commun de pouvoir bloquer indéfiniment, ce qui représente un gâchis de ressource si le thread doit être annulé. Il existe aussi un nombre important de routines qui *peuvent* être des points d'annulation, si leurs concepteurs le désirent. On consultera à cet effet la documentation Gnu pour connaître les fonctions de bibliothèques concernées.

Un grand nombre de fonctions et d'appels-système modifient temporairement des données statiques et ne peuvent pas se permettre d'être interrompus n'importe quand. Cela signifie qu'un appel-système lent, comme `read()`, ne doit jamais être invoqué si le thread est configuré avec une annulation asynchrone. Les fonctions qui supportent l'annulation asynchrone (*async-cancel safe*) sont explicitement documentées comme telles. Étant donné qu'elles sont extrêmement rares, on se fixera comme règle de ne configurer un thread en mode d'annulation asynchrone que pour des portions de code où il réalise des boucles de calculs intenses, sans aucun appel-système.

En résumé, on utilisera principalement les configurations suivantes :

Configuration	Utilisation
`PTHREAD_CANCEL_DISABLE`	Zone critique où l'on ne peut supporter aucune annulation, même si on invoque un appel-système.
`PTHREAD_CANCEL_ENABLE` `PTHREAD_CANCEL_DEFERRED`	Comportement habituel des threads.
`PTHREAD_CANCEL_ENABLE` `PTHREAD_CANCEL_ASYNCHRONOUS`	Boucle de calculs gourmande en CPU, sans appel-système.

Comme un thread peut être légitimement annulé pratiquement à n'importe quel moment, il convient de trouver un moyen de libérer les ressources qu'il peut maintenir, avant qu'il se termine vraiment. En effet, quant un thread disparaît, la mémoire dynamique qu'il s'était allouée n'est pas libérée automatiquement par le noyau, contrairement à la fin d'un processus. Le même problème se pose avec les fichiers ouverts, les tubes de communication, les sockets réseau… De plus, un thread peut avoir verrouillé une ou plusieurs ressources partagées, afin d'en avoir temporairement l'usage exclusif, et il convient de relâcher les verrous posés.

Pour cela, la norme SUSv3 propose un mécanisme assez élégant, bien qu'un peu surprenant à première vue. Lorsqu'un thread s'attribue une ressource – mémoire, fichier, verrous, etc. – qui nécessitera une libération ultérieure, il enregistre le nom d'une routine de libération dans une pile spéciale, avec la fonction **`pthread_cleanup_push()`**. Lorsque le thread se termine, les routines sont dépilées – dans l'ordre inverse de leur enregistrement – et exécutées.

Quand un thread désire libérer explicitement une ressource, à la fin d'une fonction par exemple, il appelle **pthread_cleanup_pop()**, qui extrait la dernière routine enregistrée et l'invoque. Les prototypes de ces deux fonctions sont les suivants :

```
void pthread_cleanup_push (void (* fonction) (void * argument),
                           void * argument);
void pthread_cleanup_pop (int execution_routine);
```

La routine `pthread_cleanup_push()` prend donc en premier argument un pointeur de fonction, et en second un pointeur générique pouvant représenter n'importe quel objet. Lorsque la routine de nettoyage est appelée, elle reçoit en argument le second pointeur. En général, on utilisera

```
    FILE * fp;
    fp = fopen(nom_fichier, "r");
    pthread_cleanup_push(fclose, fp);
```

ou

```
    char * buffer;
    buffer = malloc(BUFSIZE);
    pthread_cleanup_push(free, buffer);
```

ou encore

```
    int   fd;
    fd = open(nom_fichier, O_RDONLY);
    pthread_cleanup_push(close, (void *) fd);
```

Lorsqu'on désire invoquer explicitement la routine de libération, on emploie `pthread_cleanup_pop()` en lui passant un argument entier. Si cet argument est nul, la routine est retirée de la pile de nettoyage, mais elle n'est pas exécutée. Sinon, la routine est extraite et invoquée.

Ce mécanisme nécessite donc de soigner la conception du programme pour disposer les allocations et libérations autour des zones où le thread peut être annulé. On se disciplinera pour adopter ce comportement dans toutes les fonctions de l'application. En employant par exemple :

```
    void
routine_dialogue (char * nom_serveur, char *nom_fichier_enregistrement)
{
    char * buffer;
    FILE * fichier;
    int    socket_serveur;
    int    nb_octets_recus;

    buffer = malloc(BUFSIZE);
    if (buffer != NULL) {
        pthread_cleanup_push(free, buffer);

        socket_serveur = ouverture_socket(nom_serveur);
        if (socket_serveur >= 0) {
            pthread_cleanup_push(close, (void *) socket_serveur);

            fichier = fopen(nom_fichier_enregistrement, "w");
            if (fichier != NULL) {
                pthread_cleanup_push(fclose, fichier);
```

```
                while (1) {
                    nb_octets_recus = lecture_socket(socket_serveur,
                                                     buffer);
                    if (nb_octets_recus < 0)
                        break;
                    if (fwrite(buffer, 1, nb_octets_recus, fichier)
                              != nb_octets_recus)
                        break;
                }
                pthread_cleanup_pop(1); /* fclose(fichier) */
            }
            pthread_cleanup_pop(1); /* close(socket_serveur) */
        }
        pthread_cleanup_pop(1); /* free(buffer) */
    }
}
```

Pour obliger le programmeur à adopter un comportement cohérent dans toute la fonction, la norme SUSv3 impose une contrainte assez restrictive à l'utilisation de ces routines. En effet, les appels `pthread_cleanup_push()` et `pthread_cleanup_pop()` doivent se trouver dans la même fonction et dans le même bloc lexical. Cela signifie qu'ils doivent être au même niveau d'imbrication entre accolades. On peut le vérifier d'un coup d'œil en s'assurant que le `pthread_cleanup_pop()` se trouve bien au même niveau d'indentation que le `pthread_cleanup_push()` correspondant.

Pour comprendre la raison de cette restriction, il suffit de savoir que l'implémentation de ces routines dans la plupart des bibliothèques, dont NPTL, est réalisée par deux macros : la première comprend une accolade ouvrante alors que la seconde contient l'accolade fermante associée. Il importe donc de considérer ces deux fonctions comme une paire d'accolades et de ne pas essayer de les séparer de plus d'un bloc lexical.

On prendra comme habitude de faire systématiquement suivre un appel de la fonction `pthread_cleanup_pop()` d'un commentaire indiquant son effet, comme on peut le voir ci-dessus. On remarquera également que le fait de n'avoir plus qu'un seul point de sortie d'une routine oblige parfois à une indentation excessive. Pour éviter ce problème, on peut scinder la routine en plusieurs sous-fonctions ou utiliser des sauts `goto`, comme c'est souvent l'usage pour les gestions d'erreur. Dans ce cas, le programme précédent deviendrait :

```
    void
routine_dialogue (char * nom_serveur, char *nom_fichier_enregistrement)
{
    char * buffer;
    FILE * fichier;
    int    socket_serveur;
    int    nb_octets_recus;

    if ((buffer = malloc(BUFSIZE))== NULL)
        return;
    pthread_cleanup_push(free, buffer);

    if ((socket_serveur = ouverture_socket(nom_serveur)) < 0)
        goto sortie_cleanup_1;
    pthread_cleanup_push(close, (void *)socket_serveur);
```

```
    if ((fichier = fopen(nom_fichier_enregistrement, "w")) == NULL)
        goto sortie_cleanup_2;
    pthread_cleanup_push(fclose, fichier);

    while (1) {
        nb_octets_recus = lecture_socket(socket_serveur, buffer);
        if (nb_octets_recus < 0)
            break;
        if (fwrite(buffer, 1, nb_octets_recus, fichier)
              != nb_octets_recus)
            break;
    }

    pthread_cleanup_pop(1); /* fclose(fichier) */
sortie_cleanup_2 :
    pthread_cleanup_pop(1); /* close(socket_serveur) */
sortie_cleanup_1 :
    pthread_cleanup_pop(1); /* free(buffer) */
}
```

Ce genre de code est peut-être moins élégant, mais il est particulièrement bien adapté à ce type de gestion d'erreur. On rencontre de fréquents exemples d'emploi de `goto` dans ce contexte au sein des sources du noyau Linux.

À l'opposé des routines de nettoyage en fin de thread, on a souvent besoin, dans un module d'une application, d'initialiser des données au début de leur mise en œuvre. Imaginons un module servant à interroger une base de données. Il doit dissimuler au reste du programme l'implémentation interne. Que la base de données soit représentée par un fichier local, un démon fonctionnant en arrière-plan ou un serveur distant accessible par le réseau, le module doit adopter la même interface vis-à-vis du reste du programme. Il existe ainsi une routine publique d'interrogation susceptible d'être appelée par différents threads, éventuellement de manière concurrente, gérant donc l'accès critique aux données partagées avec des mécanismes décrits dans les prochains paragraphes.

Toutefois il est nécessaire, lors de la première interrogation, d'établir la liaison avec la base de données proprement dite – ouvrir le fichier, accéder au tube de communication avec le démon, contacter le serveur distant –, ce qui ne doit être réalisé qu'une seule fois. On vérifiera donc à chaque interrogation si l'initialisation a bien eu lieu. La bibliothèque Pthreads propose une fonction **`pthread_once()`** qui remplit ce rôle en s'affranchissant des problèmes de synchronisation si plusieurs threads l'invoquent simultanément.

Pour utiliser cette fonction, il faut préalablement définir une variable statique de type **`pthread_once_t`**, initialisée avec la constante `PTHREAD_ONCE_INIT`, qu'on passera par adresse à la routine `pthread_once()`. Le second argument est un pointeur sur une fonction d'initialisation qui ne sera ainsi appelée qu'une seule fois dans l'application.

```
int pthread_once (pthread_once_t * once, void (* fonction) (void));
```

En poursuivant notre exemple concernant un module de dialogue avec une base de données, on obtiendrait :

```
    void    initialisation_dialogue_base (void);
```

```
    int
interrogation_base_donnees (char * question)
{
    static pthread_once_t    once_initialisation = PTHREAD_ONCE_INIT;

    pthread_once(& once_initialisation, initialisation_dialogue_base);
     [...]
}
```

Naturellement, seule la première invocation de `pthread_once()` a un effet, les appels ultérieurs n'ayant plus aucune influence.

On peut se demander ce qui se passe lorsqu'un thread appelle `fork()`. Le comportement est tout à fait logique. Le processus entier est dupliqué, y compris les zones de mémoire partagées avec les autres threads. Par contre, il n'y a dans le processus fils qu'un seul fil d'exécution, celui du thread qui a invoqué `fork()`, cela quel que soit le nombre de threads concurrents avant la séparation.

Un premier problème se pose, car les piles et les zones de mémoire dynamiquement allouées par les autres threads continuent d'être présentes dans l'espace mémoire du nouveau processus, même s'il n'a aucun moyen d'y accéder. Aussi, en théorie ce mécanisme doit être restreint uniquement à l'utilisation de `exec()` après le `fork()`.

Un second problème peut se poser si un autre thread a verrouillé – dans le processus père – une ressource. Si le thread restant dans le processus fils a besoin de cette ressource, celle-ci persiste à être verrouillée, et on risque un blocage définitif.

Pour résoudre ce problème, il existe une fonction nommée **`pthread_atfork()`**, qui permet d'enregistrer des routines qui seront automatiquement invoquées si un thread appelle `fork()`. Les fonctions sont exécutées dans l'ordre inverse de leur enregistrement, comme avec une pile. La liste des fonctions mémorisées est commune à tous les threads.

On peut enregistrer trois fonctions avec `pthread_atfork()`. La première routine est appelée avant le `fork()` dans le thread qui l'invoque. Les deux autres routines sont appelées après la séparation, l'une dans le processus fils, et l'autre dans le processus père – toujours au sein du thread ayant invoqué `fork()`.

```
int pthread_atfork (void (* avant) (void),
                    void (* dans_pere) (void),
                    void (* dans_fils) (void));
```

Si un pointeur est nul, la routine est ignorée. Nous allons voir dans le prochain paragraphe comment bloquer ou libérer des verrous pour l'accès à des zones critiques. Dans l'encadrement de `fork()`, on essaye d'éviter la situation suivante :

1. Le thread numéro 1 bloque un verrou pour accéder à une zone de données.
2. Le thread numéro 2 appelle `fork()`, dupliquant l'ensemble de l'espace mémoire du processus, y compris le verrou bloqué.
3. Le processus père continue de se dérouler normalement, le thread 1 libérant le verrou après ses modifications, et le thread 2 pouvant poursuivre son exécution.
4. Dans le processus fils, le thread 2 veut accéder à la zone de données commune. Celle-ci étant verrouillée, il attend que le thread 1 libère le verrou, mais il n'y a pas de thread 1 dans le fils ! Le processus est définitivement bloqué.

La bonne manière de procéder est la suivante, un peu complexe mais correcte :

1. Avant que le thread 1 bloque le verrou – par exemple pendant son initialisation –, on installe la routine `avant()`, qui correspond à un blocage du verrou, ainsi que `dans_pere()` et `dans_fils()`, deux routines qui représentent une libération du verrou.
2. Le thread numéro 1 bloque le verrou.
3. Le thread numéro 2 appelle `fork()`. La routine `avant()` est invoquée. Correspondant à une demande de blocage du verrou, elle reste bloquée jusqu'à ce que le thread 1 ait terminé son travail.
4. Le thread numéro 1 libère le verrou.
5. La routine `avant()` bloque le verrou et se termine.
6. L'appel-système `fork()` a lieu, les processus se séparent. Les routines `dans_pere()` et `dans_fils()` sont invoquées, libérant le verrou dans les deux contextes.
7. Les threads 1 et 2 du processus père continuent normalement.
8. Le thread 2 du processus fils peut accéder à la zone de données s'il le désire, le verrou est libre.

Nous voyons qu'il faut donc enregistrer une série de routines pour chaque verrou susceptible d'être employé dans le processus fils, ce qui complique – parfois excessivement – l'écriture des programmes.

En fait, l'appel `pthread_atfork()` est principalement employé dans des programmes expérimentaux, pour étudier justement les blocages dus aux partages de verrous. Dans des applications courantes, on évite généralement de se trouver dans cette situation. Pour cela, on essaye de ne pas utiliser `fork()`, ou de le faire suivre immédiatement d'un `exec()`. On peut aussi appeler `fork()` avant la création des threads et installer un mécanisme de communication entre processus, comme nous en verrons au chapitre 28.

Zones d'exclusions mutuelles

L'un des enjeux essentiels lors du développement d'applications multithreads est la synchronisation entre les différents fils d'exécution concurrents. Ce qui représente, somme toute, un aspect annexe des logiciels reposant sur plusieurs processus devient ici un point crucial. Les différents threads d'une application disposant d'un accès partagé immédiat à toutes les variables globales, descripteurs de fichiers, etc., leur synchronisation est indispensable pour éviter la corruption de données et les situations de blocage.

Il existe essentiellement deux cas où des données risquent d'être corrompues si l'accès aux ressources communes n'est pas synchronisé :

- Deux threads concurrents veulent modifier une variable globale, par exemple décrémenter un compteur dans une gestion de stocks. Le premier thread lit la valeur initiale V_0 dans un registre du processeur. Il décrémente la valeur d'une unité. L'ordonnanceur commute les tâches et donne la main au second thread. Celui-ci lit la valeur initiale V_0, la décrémente et écrit la nouvelle valeur $V_0 - 1$ dans le compteur. L'ordonnanceur réactive le premier thread qui inscrit à son tour la valeur calculée $V_0 - 1$ dans le compteur. Au final, le stock indique $V_0 - 1$ unités alors qu'il aurait dû être décrémenté deux fois. Ceci présage de sérieux problèmes le jour de l'inventaire…

- Un thread modifie une structure de données globale tandis qu'un autre essaye de la lire. Le thread lecteur charge les premiers membres de la structure. L'ordonnanceur bascule le contrôle au thread écrivain, qui modifie toute la structure. Lorsque le second thread est réactivé, il lit la fin de la structure. Les premiers membres qu'il a reçus ne sont pas cohérents avec les suivants. Le problème pourrait être le même avec une chaîne de caractères, ou même une simple variable de type réel ou entier long.

Pour accéder à des données globales, il est donc indispensable de mettre en œuvre un mécanisme d'exclusion mutuelle des threads. Ce principe repose sur des données appelées *mutex*, de type **pthread_mutex_t**. Chaque variable sert de verrou pour l'accès à une zone particulière de la mémoire globale.

Il existe deux états pour un mutex : disponible ou verrouillé. Lorsqu'un mutex est verrouillé par un thread, on dit que ce dernier *tient* le mutex. Un mutex ne peut être tenu que par un seul thread à la fois. En conséquence, il existe essentiellement deux fonctions de manipulation des mutex : une fonction de verrouillage et une fonction de libération. Lorsqu'un thread demande à verrouiller un mutex déjà maintenu par un autre thread, le premier est bloqué jusqu'à ce que le mutex soit libéré.

On peut initialiser un mutex de manière statique ou dynamique, en précisant certains attributs à l'aide d'un objet de type **phtread_mutexattr_t**. L'initialisation statique se fait à l'aide de la constante **PTHREAD_MUTEX_INITIALIZER** :

```
pthread_mutex_t mutex = PHTREAD_MUTEX_INITIALIZER;
```

Pour l'initialisation dynamique, on emploie **pthread_mutex_init()** avec une variable regroupant les attributs du mutex.

```
int pthread_mutex_init (pthread_mutex_t * mutex,
                        const pthread_mutexattr_t * attributs);
```

On l'emploie généralement ainsi :

```
pthread_mutex_t      mutex;
pthread_mutexattr_t  mutexattr.

    /* initialisation de mutexattr */
    [...]
    /* initialisation du mutex     */
    if ((mutex = malloc(sizeof(pthread_mutex_t)) == NULL)
        return -1;
    pthread_mutex_init(& mutex, & mutexattr);
    [...]
```

Étant donné que les mutex servent à synchroniser différents threads, on les déclare naturellement dans des variables globales ou dans des variables locales statiques.

On peut utiliser un pointeur `NULL` en second argument de `pthread_mutex_init()` si le mutex doit avoir les attributs par défaut. Nous verrons plus bas comment configurer les attributs désirés. Une fois qu'un mutex n'est plus utilisé, on libère la variable en appelant **pthread_mutex_destroy()**.

Le mutex ne doit plus être verrouillé, sinon cette fonction échoue avec l'erreur EBUSY.

```
int pthread_mutex_destroy (pthread_mutex_t * mutex);
```

La fonction de verrouillage s'appelle **pthread_mutex_lock()**. Si le mutex est libre, il est immédiatement verrouillé et attribué au thread appelant. Si le mutex est déjà maintenu par un autre thread, la fonction reste bloquée jusqu'à la libération du mutex, puis elle le verrouille à la disposition du thread appelant. Cette fonction peut donc rester bloquée indéfiniment. Ce n'est pourtant pas un point d'annulation car la norme SUSv3 réclame que l'état des mutex soit parfaitement prévisible lors de l'annulation d'un thread. Si pthread_mutex_lock() pouvait être un point d'annulation, l'état du thread serait imprévisible.

Si pthread_mutex_lock() est invoquée sur un mutex déjà maintenu par le thread appelant, le résultat dépend du type de mutex – déterminé par les attributs employés lors de l'initialisation :

- Un mutex normal bloque le thread appelant jusqu'à sa libération. Comme celle-ci est impossible, le thread reste bloqué définitivement.
- Si le mutex est de type *récursif* – extension non portable – le thread le verrouille à nouveau en incrémentant un compteur interne. Il faudra alors débloquer le mutex un nombre égal de fois pour qu'il devienne vraiment disponible.
- Si nous avons à faire à un mutex de diagnostic – également non portable –, la fonction pthread_mutex_lock() échoue en renvoyant le code EDEADLOCK qui indique une situation de blocage définitif. Cela permet de rechercher les cas d'erreur lors d'une session de débogage.

Le prototype de pthread_mutex_lock() est le suivant :

```
int pthread_mutex_lock (pthread_mutex_t * mutex);
```

La libération d'un mutex se fait avec la fonction **pthread_mutex_unlock()**. Si le mutex est récursif, il ne sera effectivement débloqué que si le compteur interne de verrouillage tombe à zéro. Avec un mutex de diagnostic, une erreur (EPERM) se produit si le thread appelant ne possède pas le mutex. Avec les autres mutex, cette vérification n'a pas lieu, mais ce comportement n'est pas standard.

```
int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

Enfin, il existe une fonction nommée **pthread_mutex_trylock()** fonctionnant comme pthread_mutex_lock(), à la différence qu'elle échoue avec l'erreur EBUSY, plutôt que de rester bloquée, si le mutex est déjà verrouillé.

```
int pthread_mutex_trylock (pthread_mutex_t * mutex);
```

Il est généralement déconseillé d'employer pthread_mutex_trylock(). Notamment, si on désire surveiller plusieurs mutex à la fois, on n'utilisera **pas** une construction du genre :

```
    while (1) {
        if (pthread_mutex_trylock(& mutex_1) == 0)
            break;
        if (pthread_mutex_trylock(& mutex_2) == 0)
            break;
        if (pthread_mutex_trylock(& mutex_3) == 0)
            break;
    }
```

Ce code est très mauvais car il gâche inutilement des ressources CPU, alors qu'il est possible de le remplacer par une attente de conditions, comme nous le verrons dans la prochaine section.

Les attributs d'un mutex, enregistrés dans un objet de type `mutex_attr_t`, peuvent être initialisés avec la fonction **`pthread_mutexattr_init()`** et détruits avec **`pthread_mutexattr_destroy()`**.

```
int pthread_mutexattr_init (pthread_mutexattr_t * attributs);
int pthread_mutexattr_destroy (pthread_mutexattr_t * attributs);
```

Les variables `pthread_mutexattr_t`, avec la bibliothèque NPTL, ne comportent qu'un seul attribut, le type de mutex. Pour le configurer, on emploie la fonction **`pthread_mutexattr_settype()`** et **`pthread_mutexattr_gettype()`** pour le lire.

```
int pthread_mutexattr_settype (pthread_mutexattr_t * attributs,
                               int type);
int pthread_mutexattr_gettype (pthread_mutexattr_t * attributs,
                               int * type);
```

Le type d'un mutex est représenté par l'une des constantes suivantes :

Nom	Signification
`PTHREAD_MUTEX_NORMAL`	Mutex normal, rapide. L'invocation double de `pthread_mutex_lock()` dans le même thread conduit à un blocage définitif.
`PTHREAD_MUTEX_RECURSIVE`	Mutex récursif. Un même thread peut le verrouiller à plusieurs reprises, il faudra le libérer autant de fois.
`PTHREAD_MUTEX_ERRORCHECK`	Mutex de diagnostic. Une tentative de double verrouillage échoue.

Attention

Ces constantes ayant été normalisées par SUSv3 alors qu'elles étaient absentes de la norme Posix.1c précédente, il faut initialiser la constante symbolique `_XOPEN_SOURCE` avec la valeur `500` avant d'inclure `<pthread.h>`.

Le programme suivant utilise un mutex comme verrou pour restreindre l'accès au flux `stdout`. Nous lançons en parallèle une dizaine de threads, qui vont attendre une durée aléatoire avant de demander un blocage du mutex. L'attente aléatoire sert à perturber un peu le déterminisme de l'ordonnanceur et à éviter de voir les threads se dérouler dans l'ordre croissant.

exemple_mutex.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

static void *  routine_threads (void * argument);
static int     aleatoire       (int maximum);
```

```
    pthread_mutex_t   mutex_stdout = PTHREAD_MUTEX_INITIALIZER;

    int
main (void)
{
    int       i;
    pthread_t thread;

    for (i = 0; i < 10; i ++)
        pthread_create(& thread, NULL, routine_threads, (void *) i);
    pthread_exit(NULL);
}

    static void *
routine_threads (void * argument)
{
    int numero = (int) argument;
    int nombre_iterations;
    int i;
    nombre_iterations = 1+aleatoire(3);
    for (i = 0; i < nombre_iterations; i ++) {
        sleep(aleatoire(3));
        pthread_mutex_lock(& mutex_stdout);
        fprintf(stdout, "Le thread %d a obtenu le mutex \n", numero);
        sleep(aleatoire(3));
        fprintf(stdout, "Le thread %d relâche le mutex \n", numero);
        pthread_mutex_unlock(& mutex_stdout);
    }
    return NULL;
}

    static int
aleatoire (int maximum)
{
    double d;
    d = (double) maximum * rand();
    d = d / (RAND_MAX + 1.0);
    return ((int) d);
}
```

On remarque l'emploi de `pthread_exit()` en fin de fonction `main()` pour terminer le fil d'exécution principal, sans finir les autres threads. Ceci est parfaitement défini dans la norme SUSv3.

Attention

Les premières implémentations de la NPTL contiennent un bogue qui termine toute l'application si le thread *main* invoque `pthread_exit()` avant que les autres threads n'aient réellement démarré. Une solution simple pour éviter ce problème (corrigé dans les implémentations ultérieures) est d'ajouter un petit sommeil d'une seconde avant le `pthread_exit()`.

Le déroulement du processus montre bien que l'accès est correct, malgré les demandes concurrentes de verrouillage du mutex.

```
$ ./exemple_mutex
Le thread 2 a obtenu le mutex
Le thread 2 relâche le mutex
Le thread 2 a obtenu le mutex
Le thread 2 relâche le mutex
Le thread 4 a obtenu le mutex
Le thread 4 relâche le mutex
Le thread 5 a obtenu le mutex
Le thread 5 relâche le mutex
Le thread 5 a obtenu le mutex
Le thread 5 relâche le mutex
Le thread 8 a obtenu le mutex
Le thread 8 relâche le mutex
Le thread 9 a obtenu le mutex
Le thread 9 relâche le mutex
Le thread 0 a obtenu le mutex
Le thread 0 relâche le mutex
Le thread 7 a obtenu le mutex
Le thread 7 relâche le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relâche le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relâche le mutex
Le thread 6 a obtenu le mutex
Le thread 6 relâche le mutex
Le thread 4 a obtenu le mutex
Le thread 4 relâche le mutex
Le thread 2 a obtenu le mutex
Le thread 2 relâche le mutex
Le thread 8 a obtenu le mutex
Le thread 8 relâche le mutex
Le thread 9 a obtenu le mutex
Le thread 9 relâche le mutex
Le thread 0 a obtenu le mutex
Le thread 0 relâche le mutex
Le thread 7 a obtenu le mutex
Le thread 7 relâche le mutex
Le thread 7 a obtenu le mutex
Le thread 7 relâche le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relâche le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relâche le mutex
Le thread 6 a obtenu le mutex
Le thread 6 relâche le mutex
Le thread 8 a obtenu le mutex
Le thread 8 relâche le mutex
Le thread 0 a obtenu le mutex
Le thread 0 relâche le mutex
```

```
Le thread 1 a obtenu le mutex
Le thread 1 relâche le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relâche le mutex
$
```

La définition des verrous corrects à employer pour accéder aux données partagées est une tâche importante de la conception des programmes multithreads. En prenant l'exemple d'une grosse base de données – des réservations ferroviaires par exemple –, il serait vraiment peu efficace de verrouiller l'ensemble de la base à chaque fois qu'un thread veut ajouter un enregistrement. D'un autre côté, un trop grand nombre de mutex indépendants peut aussi devenir problématique. Qu'un thread ait systématiquement besoin de verrouiller simultanément plusieurs mutex peut être très dangereux, car la moindre maladresse dans le programme risque de déclencher des blocages irrémédiables. Dans cette situation d'étreinte fatale, un thread maintient un mutex et en attend un autre, alors qu'un autre thread est coincé dans la situation inverse.

Il est donc indispensable de bien dimensionner le problème et de décider de la granularité des portions protégées par un mutex.

Attente de conditions

Lorsqu'un processus doit attendre le déblocage du premier mutex disponible dans un ensemble, ou s'il doit patienter jusqu'à ce qu'un événement survienne dans un autre thread, on emploie une autre technique de synchronisation. Il existe des variables « conditions » représentées par le type `pthread_cond_t`. Un thread peut se mettre en attente d'une condition, et lorsqu'elle est réalisée par un autre thread, ce dernier l'en avertit directement.

Le principe est simple, reposant sur deux fonctions de manipulation des conditions : l'une est l'attente de la condition, le thread appelant restant bloqué jusqu'à ce qu'elle soit réalisée, et l'autre sert à signaler que la condition est remplie. C'est l'application qui affecte une signification à la variable condition, qui est simplement considérée comme une variable booléenne un peu spéciale par la bibliothèque Pthreads.

Les variables conditions ont des attributs, de type `pthread_condattr_t`, qui n'ont pas d'utilité dans la bibliothèque NPTL. En conséquence, on initialisera généralement les conditions de manière statique

```
pthread_cond_t   condition  = PTHREAD_COND_INITIALIZER;
```

ou en employant la fonction **`pthread_cond_init()`**, en passant un second argument `NULL`

```
int pthread_cond_init (pthread_cond_t * condition,
                       pthread_condattr_t * attributs);
```

Une condition inutilisée est libérée avec **`pthread_cond_destroy()`**. Aucun autre thread ne doit être en attente sur la condition, sinon la libération échoue avec l'erreur `EBUSY`.

```
int pthread_cond_destroy (pthread_cond_t * condition);
```

Voyons à présent l'utilisation effective d'une condition. Tout d'abord, il faut signaler qu'une condition est toujours associée à un mutex, ceci pour éviter des problèmes de concurrence d'accès sur la variable.

Examinons d'abord le thread qui doit attendre une condition :

1. On initialise la variable condition et le mutex qui lui est associé.
2. Le thread bloque le mutex. Ensuite, il invoque la routine **`pthread_cond_wait()`** qui attend que la condition soit réalisée.
3. Le thread libère le mutex.

Maintenant, voyons le thread qui réalise la condition :

1. Le thread travaille jusqu'à avoir réalisé la condition attendue.
2. Il bloque le mutex associé à la condition.
3. Le thread appelle la fonction **`pthread_cond_signal()`** pour montrer que la condition est remplie.
4. Le thread débloque le mutex.

Ce schéma est *a priori* surprenant puisqu'il semble que, lorsque le second thread désire signaler la réalisation de la condition, l'accès lui soit interdit, le premier thread ayant bloqué le mutex. En fait, la fonction `pthread_cond_wait()` fonctionne en trois temps :

1. D'abord, elle débloque le mutex associé à la condition, et elle se met en attente. Cette opération est réalisée de manière atomique vis-à-vis de la bibliothèque Pthreads.
2. L'attente se poursuit jusqu'à ce que la réalisation de la condition soit indiquée.
3. La condition étant remplie, la fonction termine son attente et bloque à nouveau le mutex, avant de revenir dans le programme appelant.

Le scénario se déroule donc ainsi :

Thread attendant la condition	Thread signalant la condition
Appel de `pthread_mutex_lock()` : blocage du mutex associé à la condition.	
Appel de `pthread_cond_wait()` : déblocage du mutex.	
… attente …	
	Appel de `pthread_mutex_lock()` sur le mutex.
	Appel de `pthread_cond_signal()`, qui réveille l'autre thread.
Dans `pthread_cond_wait()`, tentative de récupérer le mutex. Blocage.	
	Appel de `pthread_mutex_unlock()`. Le mutex étant libéré, l'autre thread se débloque.
Fin de `pthread_cond_wait()`.	
Appel de `pthread_mutex_unlock()` pour revenir à l'état initial.	

On peut vérifier qu'il n'y a pas de risque d'interblocage des deux threads ni de risque de perdre la signalisation d'une condition dès que le premier `pthread_mutex_lock()` a été invoqué.

Les prototypes de ces deux fonctions sont les suivants :

```
int pthread_cond_signal (pthread_cond_t * condition);
int pthread_cond_wait   (pthread_cond_t * condition,
                         pthread_mutex_t * mutex);
```

Le terme « signal » présent dans `pthread_cond_signal()` ne doit pas être confondu avec les signaux que nous avons étudiés dans les chapitres 6 à 8. Nous reviendrons sur les interactions entre threads et signaux à la fin du chapitre.

Dans l'exemple suivant, un thread sert à gérer des alarmes, alors qu'un autre surveille (simule) des variations de température.

exemple_condition.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <pthread.h>

    pthread_cond_t   condition_alarme
                        = PTHREAD_COND_INITIALIZER;
    pthread_mutex_t  mutex_alarme
                        = PTHREAD_MUTEX_INITIALIZER;

    static void *   thread_temperature (void * inutile);
    static void *   thread_alarme      (void * inutile);
    static int      aleatoire          (int    maximum);

    int
main (void)
{
    pthread_t thr;
    pthread_create(& thr, NULL, thread_temperature, NULL);
    pthread_create(& thr, NULL, thread_alarme,      NULL);
    pthread_exit(NULL);
}

    static void *
thread_temperature (void * inutile)
{
    int  temperature = 20;
    while (1) {
        temperature += aleatoire(5) - 2;
        fprintf(stdout, "Température : %d \n", temperature);
        if ((temperature < 16) || (temperature > 24)) {
            pthread_mutex_lock(& mutex_alarme);
            pthread_cond_signal(& condition_alarme);
            pthread_mutex_unlock(& mutex_alarme);
        }
        sleep(1);
    }
    return NULL;
}
```

```
    static void *
thread_alarme (void * inutile)
{
    while (1) {
        pthread_mutex_lock(& mutex_alarme);
        pthread_cond_wait(& condition_alarme,
                          & mutex_alarme);
        pthread_mutex_unlock(& mutex_alarme);
        fprintf(stdout, "ALARME\n");
    }
    return NULL;
}
```

L'exécution montre bien l'activation du thread d'alarme lorsque la condition est signalée.

```
$ ./exemple_condition
Température : 22
Température : 21
Température : 22
Température : 23
Température : 25
ALARME
Température : 23
Température : 22
Température : 23
Température : 22
Température : 22
Température : 23
Température : 22
Température : 22
Température : 24
Température : 26
ALARME
Température : 27
ALARME
  (Contrôle-C)
$
```

En fait, ce programme peut poser un problème. La norme SUSv3 autorise l'appel `pthread_cond_wait()` à se terminer de manière impromptue, même si la condition n'est pas réalisée. Ceci, je suppose, permet de simplifier l'implémentation d'une bibliothèque Pthreads vis-à-vis des appels-système lents interrompus par un signal.

Il faut donc accompagner l'invocation de `pthread_cond_wait()` d'une vérification de l'état de la condition. Dans notre cas, cela nécessiterait d'une part de transférer la variable `temperature` en zone globale partagée et d'utiliser un mutex – éventuellement le même – pour accéder à son contenu. On aurait donc quelque chose comme :

```
pthread_mutex_lock (& mutex_alarme);
while ((temperature > 15) && (temperature < 25))
    pthread_cond_wait (& condition_alarme, & mutex_alarme);
    pthread_mutex_unlock (& mutex_alarme);
```

D'autre part, la modification de la variable `temperature` dans le second thread devrait être encadrée par un couple blocage / déblocage de `mutex_alarme`.

Il existe une fonction d'attente temporisée, nommée **`pthread_cond_timedwait()`**, permettant de limiter le délai imparti pour la réalisation de la condition.

```
int pthread_cond_timedwait (pthread_cond_t * condition,
                            pthread_mutex_t * mutex,
                            const struct timespec * date);
```

Attention, on ne précise pas la durée d'attente mais l'heure maximale jusqu'à laquelle la fonction peut attendre. La structure `timespec` a été décrite dans le chapitre 8, elle contient un champ contenant le nombre de secondes écoulées depuis le 1er janvier 1970, et un champ indiquant le complément en nanosecondes. Pour obtenir la date actuelle, on peut employer les appels-système `time()` ou `gettimeofday()`, que nous étudierons dans le chapitre 25.

Si le délai est dépassé, cette fonction échoue avec l'erreur `ETIMEDOUT`. Même dans ce cas, il est important de s'assurer si la condition n'est pas vérifiée quand même, notamment si plusieurs threads attendent la réalisation de la même condition.

Dans ce cas en effet, la fonction `pthread_cond_signal()` garantit qu'un seul thread en attente sera réveillé. Lorsqu'on désire réveiller tous les threads qui surveillent cette condition, il faut employer **`pthread_cond_broadcast()`**. Dans un cas comme dans l'autre, aucune erreur ne se produit si aucun thread n'est en attente.

```
int pthread_cond_broadcast (pthread_cond_t * condition);
```

Les personnes découvrant la programmation multithread sont souvent surprises par le comportement de `pthread_cond_wait()` comme point d'annulation. En effet, lorsqu'un thread reçoit une demande d'annulation durant cette fonction d'attente, elle se termine, mais doit récupérer d'abord le mutex associé à la condition. Cela signifie qu'elle peut bloquer indéfiniment avant de se terminer.

Cette attitude peut surprendre si on considère l'annulation comme une demande de terminaison urgente, à la manière d'un signal `SIGQUIT`. Mais ce n'est pas la bonne façon de voir cette fonctionnalité. Il est préférable d'imaginer la demande d'annulation à la manière des applications graphiques dans lesquelles le clic sur un bouton Fermeture ne termine pas nécessairement l'application mais peut passer par une phase de sauvegarde éventuelle des données modifiées si l'utilisateur le désire.

L'annulation d'un thread doit laisser les données manipulées dans un état prévisible, et le seul état prévisible du mutex associé à un appel `pthread_cond_wait()` est le verrouillage. Bien entendu, le thread ne doit pas se terminer en laissant le mutex bloqué. Il faut donc utiliser une fonction de nettoyage :

```
pthread_mutex_lock(& mutex);
pthread_cleanup_push(pthread_mutex_unlock, (void *) & mutex);
while (! condition_realisee)
   pthread_cond_wait(& condition, & mutex);
   pthread_cleanup_pop(1); /* pthread_mutex_unlock (& mutex) */
```

Pour terminer cette section sur les variables conditions, mentionnons qu'il existe deux fonctions **`pthread_condattr_init()`** et **`pthread_condattr_destroy()`** permettant de manipuler les attributs des conditions de manière dynamique. Ceci n'a pas grand intérêt sous Linux car l'implémentation de la bibliothèque NPTL ne gère aucun attribut pour les variables conditions :

```
int pthread_condattr_init (pthread_condattr_t * attributs);
int pthead_condattr_destroy (pthread_condattr_t * attributs);
```

Sémaphores

La bibliothèque NPTL implémente un mécanisme de synchronisation qui a été introduit par la norme Posix.1b (temps-réel) : les *sémaphores*. Ces fonctionnalités sont déclarées dans `<semaphore.h>`, elles sont disponibles si la constante symbolique **`_POSIX_SEMAPHORES`** est définie dans `<unistd.h>`.

> **Attention**
> Il ne faut pas confondre les sémaphores temps-réel, dont les fonctions sont préfixées par la chaine « `sem_` », et les sémaphores Système V, dont les noms commencent par « `sem` » et que nous étudierons dans le chapitre 29.

Un sémaphore est une variable de type `sem_t` servant à limiter l'accès à une portion critique de code. Il en existe deux types : le sémaphore anonyme ne peut être utilisé que dans le processus qui l'a créé (il est toutefois accessible par tous les threads du processus). Au contraire, le sémaphore nommé est accessible depuis plusieurs processus indépendants.

L'accès aux sémaphores nommés est une nouveauté de la bibliothèque NPTL. Jusque-là, seuls les sémaphores anonymes étaient utilisables sous Linux. On a toutefois une contrainte avec les premières versions de la bibliothèque NPTL : pour que les sémaphores nommés soient utilisables, il faut que le pseudo-système de fichiers `Tmpfs` soit monté sur `/dev/shm/`. On se reportera aux documentations sur le noyau Linux pour plus de détails.

Un sémaphore anonyme sera simplement déclaré ainsi :

```
sem_t semaphore;
```

puis initialisé avec la fonction **`sem_init()`** ; on le libère symétriquement en employant **`sem_destroy()`**. Comme nous l'avons déjà observé avec les conditions, il ne faut pas qu'un thread attende un sémaphore qu'on veut libérer, sinon cette fonction échoue avec l'erreur `EBUSY` dans `errno`.

```
int sem_init (sem_t * semaphore, int partage, unsigned int valeur);
int sem_destroy (sem_t * semaphore);
```

Le second argument de `sem_init()` indique si le sémaphore est réservé au processus appelant ou s'il doit être partagé entre plusieurs processus.

Le troisième argument représente la valeur initiale du sémaphore. Cette valeur est inscrite dans un compteur qui est décrémenté chaque fois qu'un thread pénètre dans la portion critique du programme, et incrémenté à chaque sortie de cette zone critique.

L'entrée dans la portion critique ne peut se faire que si le compteur est strictement positif. Ainsi, la valeur initiale du compteur représente le nombre maximal de threads simultanément tolérés dans la zone critique. La plupart du temps, nous initialiserons nos sémaphores ainsi :

```
sem_t    semaphore;
[...]
sem_init (& semaphore, 0, 1);
```

Pour accéder à un sémaphore nommé, on utilise la fonction :

```
sem_t * sem_open (const char * nom, int options,
                  ... /* mode_t mode, unsigned int valeur */ );
```

Le principe est assez semblable à celui des fichiers. Si le nom commence par un slash, il est unique pour tous les processus y faisant référence. Les options possibles sont constituées par un OU binaire entre :

Option	Signification
O_CREAT	Création d'un nouveau sémaphore. Dans ce cas seulement les deux arguments suivants (mode et valeur) sont utilisés. Sinon ils peuvent être omis. Le troisième argument de sem_open() indique les permissions d'accès au sémaphore comme nous l'étudierons dans le chapitre 19 avec l'appel système open(). Le dernier argument est la valeur du compteur du sémaphore, comme le troisième argument de sem_init().
O_EXCL	Utilisé conjointement à l'option précédente, celle-ci garantit que l'appel système sem_open() échoue si le sémaphore existe déjà.

Lorsque l'on a terminé d'utiliser un sémaphore nommé, on peut le libérer avec :

```
int sem_close(sem_t * semaphore);
```

Lorsqu'un thread désire entrer dans la portion de code critique, il appelle la fonction **sem_wait()**, qui attend que le compteur du sémaphore soit supérieur à zéro, et le décrémente avant de revenir. La vérification de la valeur du compteur et sa décrémentation sont liées de manière atomique, évitant ainsi tout problème de concurrence d'accès. Cette fonction est un point d'annulation pour les Pthreads.

```
int sem_wait (sem_t * semaphore);
```

Une fois que le processus a fini de travailler dans la portion critique, il invoque en sortant la fonction **sem_post()**, qui incrémente le compteur.

```
int sem_post (sem_t * semaphore);
```

Cette fonction peut échouer si la valeur du compteur dépasse **SEM_VALUE_MAX**. Ceci est révélateur d'un bogue où on invoque à répétition sem_post() sans avoir appelé sem_wait() auparavant.

Il existe une fonction **sem_trywait()** fonctionnant comme sem_wait() mais qui ne bloque pas. Elle renvoie –1 et positionne EAGAIN dans errno si le compteur n'est pas supérieur à zéro.

```
int sem_trywait (sem_t * semaphore);
```

On peut aussi consulter directement la valeur du compteur d'un sémaphore en appelant **sem_getvalue()** qui stocke l'état actuel dans la variable sur laquelle on transmet un pointeur en second argument.

```
int sem_getvalue (sem_t * semaphore, int * valeur);
```

L'exemple suivant illustre une utilisation simple des sémaphores, pour limiter à trois le nombre de threads simultanément présents dans une portion critique.

exemple_semaphores.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <pthread.h>
    #include <semaphore.h>
    #include <unistd.h>

    sem_t    semaphore;

    static void * routine_thread (void * numero_thread);
    static int    aleatoire      (int maximum);

    int
main (void)
{
    int       i;
    pthread_t thread;

    sem_init(& semaphore, 0, 3);
    for (i = 0; i < 10; i ++)
        pthread_create(& thread, NULL, routine_thread, (void *) i);
    pthread_exit(NULL);
}

    void *
routine_thread (void * numero_thread)
{
    int    i;
    for (i = 0; i < 2; i ++) {
        sem_wait(& semaphore);
        fprintf(stdout, "Thread %d dans portion critique \n",
                        (int) numero_thread);
        sleep(aleatoire(4));
        fprintf(stdout, "Thread %d sort de la portion critique \n",
                         (int) numero_thread);
        sem_post(& semaphore);
        sleep(aleatoire(4));
    }
    return NULL;
}
```

L'exécution permet de vérifier la limitation du nombre de threads accédant simultanément à la portion critique.

```
$ ./exemple_semaphores
Thread 0 dans portion critique
Thread 1 dans portion critique
Thread 2 dans portion critique
Thread 1 sort de la portion critique
Thread 3 dans portion critique
Thread 0 sort de la portion critique
```

```
Thread 2 sort de la portion critique
Thread 4 dans portion critique
Thread 5 dans portion critique
Thread 3 sort de la portion critique
Thread 5 sort de la portion critique
Thread 6 dans portion critique
Thread 7 dans portion critique
Thread 7 sort de la portion critique
Thread 8 dans portion critique
Thread 4 sort de la portion critique
Thread 9 dans portion critique
Thread 6 sort de la portion critique
Thread 0 dans portion critique
Thread 9 sort de la portion critique
Thread 1 dans portion critique
Thread 0 sort de la portion critique
Thread 2 dans portion critique
Thread 8 sort de la portion critique
Thread 5 dans portion critique
Thread 2 sort de la portion critique
Thread 3 dans portion critique
Thread 3 sort de la portion critique
Thread 6 dans portion critique
Thread 1 sort de la portion critique
Thread 7 dans portion critique
Thread 5 sort de la portion critique
Thread 9 dans portion critique
Thread 6 sort de la portion critique
Thread 8 dans portion critique
Thread 7 sort de la portion critique
Thread 4 dans portion critique
Thread 9 sort de la portion critique
Thread 8 sort de la portion critique
Thread 4 sort de la portion critique
$
```

Voici un exemple simple de sémaphores nommés, partagés entre deux processus. Il s'agit d'un processus père et de son fils, mais on pourrait très bien les implémenter sous forme de deux applications indépendantes.

exemple_semaphores_nommes.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <fcntl.h>

static int processus (const char * nom);
static int aleatoire (int maximum);
```

```
        int
main (void)
{
        switch (fork()) {
                case -1:
                        perror("fork");
                        exit(EXIT_FAILURE);
                case 0:
                        return processus("Fils");

                default:
                        return processus("Père");
        }
}

        static int
processus (const char * nom)
{
        sem_t * semaphore;
        int    i;

        semaphore = sem_open("/mon_semaphore", O_CREAT, 0600, 1);
        if (semaphore == SEM_FAILED) {
                perror(nom);
                exit(EXIT_FAILURE);
        }

        for (i = 0; i < 2; i ++) {
                sem_wait(semaphore);
                fprintf(stdout, "%s dans portion critique\n", nom);
                sleep(aleatoire(4));
                fprintf(stdout, "%s sort de la portion critique\n", nom);
                sem_post(semaphore);
                sleep(aleatoire(4));
        }
        sem_close(semaphore);
        return EXIT_SUCCESS;
}
```

L'exécution nous montre que les deux processus sont bien synchronisés en ce qui concerne l'accès à la portion critique :

```
$ ./exemple_semaphores_nommes
Fils dans portion critique
Fils sort de la portion critique
Père dans portion critique
Père sort de la portion critique
Fils dans portion critique
Fils sort de la portion critique
Père dans portion critique
Père sort de la portion critique
$
```

On notera que la bibliothèque NPTL implémente également d'autres mécanismes de synchronisation, que nous ne détaillerons pas ici, mais qui sont intéressants à connaître :

- Les barrières `pthread_barrier_t` : un appel `pthread_barrier_wait()` est bloquant tant qu'un certain nombre de threads ne l'a pas invoqué. Une fois que ce nombre (programmable à l'initialisation de la barrière) est atteint, la barrière est levée et les threads sont débloqués.
- Les verrous `pthread_rwlock_t` : ils fonctionnent comme les mutex mais avec une indication qualitative de l'usage que le thread compte faire du verrou. Deux threads qui demandent le même verrou pour un accès en lecture sur les données qu'il protège seront tolérés simultanément, alors qu'un thread qui demande un verrou pour un accès en écriture devra être le seul à en disposer.

Données privées d'un thread

Les threads doivent souvent manipuler des données privées. Dans la plupart des cas, on peut simplement utiliser des variables locales qui sont stockées dans la pile privée du thread au moment de l'entrée dans la fonction. Il y a pourtant des cas où un thread a besoin d'utiliser des variables privées disponibles de manière globale. On peut par exemple imaginer un module implémentant une bibliothèque de fonctions qui stocke certaines informations dans des variables statiques entre deux invocations de fonction.

Pour permettre ce comportement, la norme SUSv3 introduit la notion de clés associées à des données privées. La clé est une variable de type **`pthread_key_t`**, qui peut résider en variable statique. La bibliothèque associe la clé avec un pointeur `void *` différent pour chaque thread.

L'initialisation d'une clé privée se fait à l'aide de la fonction **`pthread_key_create()`**, à laquelle on peut éventuellement passer un pointeur sur une fonction de destruction qui libère le pointeur associé si le thread se termine avant que la clé ne soit détruite avec **`pthread_key_delete()`**.

```
int pthread_key_create (pthread_key_t * cle_privee,
                        void (* fonction) (void *));
int pthread_key_delete (pthread_key_t cle_privee);
```

Une fois qu'une clé a été initialisée, on utilise la fonction **`pthread_setspecific()`** pour l'associer à un pointeur représentant des données personnelles du thread.

```
int pthread_setspecific (pthread_key_t cle_privee, const void * data);
```

Pour lire les données associées à une clé, on emploie **`pthread_getspecific()`**.

```
void * pthread_getspecific (pthread_key_t cle_privee);
```

En imaginant un module permettant de charger un fichier de données, puis d'accéder ensuite à son contenu à travers des fonctions d'interrogation, on peut construire le schéma suivant :

```
    pthread_key_t    cle_privee;

    int
ouverture_fichier (const char * nom_fichier);
{
    FILE * fp;
```

```
    struct donnees * donnees;
    int nb_donnees;
    int i;
    pthread_key_create(& cle_privee, free);
    fp = fopen(nom_fichier, "r");
   /* lecture nb_donnees */
    [...]
    donnees = (struct donnees *) calloc(nb_donnees,
                                        sizeof(struct donnees));
    /* lecture des données */
    for (i = 0; i < nb_donnees; i++)
        [...]
    pthread_setspecific(& cle_privee, donnees);
    return nb_donnees;
}

    int
resultat_donnee (int num)
{
    struct donnees * donnees;
    donnees = (struct donnees *) pthread_getspecific(& cle_privee);
    return donnees[num].resultats;
}
```

En fait, la robustesse du programme serait sensiblement renforcée en employant la fonction `pthread_once()` afin de garantir que l'initialisation de la clé n'ait lieu qu'une seule fois.

Le fonctionnement des données privées est assez subtil lors de la libération des clés avec `pthread_key_delete()`. Pour plus de précisions, on consultera la documentation de la bibliothèque LinuxThreads.

Une question peut se poser en ce qui concerne la variable globale `errno`, mise à jour par l'essentiel des appels-système et fonctions de bibliothèque C. En effet, on peut présumer un gros risque d'interférences si la variable globale est simultanément modifiée par des appels-système survenant dans des threads concurrents. Une solution consisterait à abandonner l'usage de cette variable et à toujours renvoyer la valeur d'erreur plutôt que d'employer –1 (comme le font d'ailleurs les routines de la bibliothèque NPTL). Néanmoins, cette méthode n'est pas applicable car elle nécessiterait de profonds bouleversements tant dans le noyau que dans la bibliothèque C. Il a donc été décidé dans la norme SUSv3 de tolérer l'existence d'une variable `errno` privée pour chaque thread.

On peut ainsi considérer qu'un thread dispose comme données privées de variables allouées dans sa pile, de données associées aux clés privées, et de la variable globale `errno`.

Les threads et les signaux

Il est généralement déconseillé d'utiliser une gestion des signaux dans les applications multithreads, mais cela est parfois indispensable. Les principes essentiels sont les suivants :

- La gestion d'un signal (en l'ignorant, en laissant le comportement par défaut, ou en installant un gestionnaire) est assurée globalement pour l'ensemble de l'application en employant la routine `sigaction()`. Cet appel-système a été décrit dans le chapitre 7.

- Le blocage temporaire d'un signal est réalisé au niveau du thread en utilisant la routine **pthread_sigmask()** qui fonctionne de manière similaire à `sigprocmask()` mais en limitant son effet au thread appelant.

```
int pthread_sigmask (int methode,
                     const sigset_t * masque,
                     sigset_t * ancien_masque);
```

- Pour envoyer un signal à un thread, on utilise **pthread_kill()**. Cette routine fonctionne comme l'appel-système `kill()`. Bien entendu, il faut que l'émission du signal se fasse de manière interne, au sein du même programme, puisqu'il faut avoir accès à la variable `pthread_t` indiquant le thread visé. On peut envoyer des signaux classiques ou des signaux temps-réel, mais dans ce cas, il faut noter que `pthread_kill()` ne permet pas d'associer une structure `siginfo` au signal, contrairement à `sigqueue()`.

```
int pthread_kill (pthread_t thread, int numero_signal);
```

- Un signal interne – émis par `pthread_kill()` - ou un signal externe synchrone – comme `SIGBUS`, `SIGFPE`, `SIGPIPE` qui viennent en réponse à une action particulière d'un thread – sont naturellement reçus par le thread visé.
- Un signal externe asynchrone doit, selon la norme SUSv3, être reçu par l'ensemble du processus, puis le noyau doit choisir arbitrairement un thread ne bloquant pas le signal pour le lui envoyer. Avant Linux 2.6, l'implémentation LinuxThreads différait légèrement de ce schéma puisque les threads étaient représentés par des processus indépendants. Il n'y avait donc pas de choix possible, le signal étant dirigé vers le thread initialement visé, même si ce dernier bloquait sa réception. Depuis le noyau 2.6, la bibliothèque NPTL permet de gérer cette situation.

Dans notre exemple, nous allons installer un gestionnaire commun pour tout un ensemble de signaux temps-réel. Ensuite chaque thread va bloquer tous ces signaux sauf un (spécifique à chaque thread). Nous enverrons depuis une autre console des signaux au processus, et verrons s'ils arrivent bien sur les threads corrects.

exemple_signaux :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <signal.h>
    #include <pthread.h>
    #include <unistd.h>

    #define NB_THREADS   5

    static void * fonction_thread (void * void_arg);
    static void   gestionnaire (int numero);

    int
main (void)
{
    int i;
    static pthread_t thr;
    struct sigaction action;
```

```
    fprintf(stdout, "Thread main, mon PID est %ld\n", (long) getpid());

    sigfillset(& action.sa_mask);
    action.sa_handler = gestionnaire;
    action.sa_flags = 0;
    for (i = SIGRTMIN; i < SIGRTMIN+NB_THREADS; i ++) {
        if (sigaction(i, & action, NULL) == 0)
            fprintf(stdout, "Signal %d capturé\n", i);
        else
            fprintf(stdout, "Signal %d NON capturé\n", i);
    }

    for (i = 0; i < 5; i ++)
        pthread_create(& thr, NULL, fonction_thread, (void *) i);

    fprintf(stdout, "Thread main, je me termine\n");
    pthread_exit(NULL);
}



    static void *
fonction_thread (void * void_arg)
{
    sigset_t  masque;
    int       numero = (int) void_arg;
    int       i;

    fprintf(stdout, "Thread %d, mon PID est %ld\n", numero,
                                (long) getpid());

    sigemptyset(& masque);
    for (i = 0; i < NB_THREADS; i ++)
        if (i != numero)
            sigaddset(& masque, SIGRTMIN + i);
    pthread_sigmask(SIG_BLOCK, & masque, NULL);

    fprintf(stdout, "Thread %d bloque tout sauf %d\n", numero,
                    SIGRTMIN+numero);

    while (1)  {
        pause();
        fprintf(stdout, "Thread %d a reçu un signal\n", numero);
    }
    return NULL;
}

    void
gestionnaire(int numero)
{
    fprintf(stdout, ">>> signal %d reçu <<<\n", numero);
}
```

L'exécution se déroule sur deux consoles différentes. Nous voyons bien qu'avec la bibliothèque NPTL les threads ont tous le même PID (contrairement à la bibliothèque LinuxThreads).

```
$ ./exemple_signaux
Thread main, mon PID est 6641
Signal 33 capturé
Signal 34 capturé
Signal 35 capturé
Signal 36 capturé
Signal 37 capturé
Thread 0, mon PID est 6641
Thread 0 bloque tout sauf 33
Thread 1, mon PID est 6641
Thread 1 bloque tout sauf 34
Thread 2, mon PID est 6641
Thread 2 bloque tout sauf 35
Thread 3, mon PID est 6641
Thread 3 bloque tout sauf 36
Thread 4, mon PID est 6641
Thread 4 bloque tout sauf 37
Thread main, je me termine
                                $ kill -33 6641
>>> signal 33 reçu <<<
Thread 0 a reçu un signal
                                $ kill -35 6641
>>> signal 35 reçu <<<
Thread 2 a reçu un signal
                                $ kill -37 6641
>>> signal 37 reçu <<<
Thread 4 a reçu un signal
                                $ kill -34 6641
>>> signal 34 reçu <<<
Thread 1 a reçu un signal
                                $ kill -36 6641
>>> signal 36 reçu <<<
Thread 3 a reçu un signal
                                $ kill -KILL 6641
Processus arrêté
$
```

Si on déconseille en règle générale d'utiliser trop de signaux dans une application multithread, c'est principalement parce que les fonctions permettant de manipuler les threads ne doivent pas être appelées depuis un gestionnaire de signaux. La norme SUSv3 indique que ces routines ne sont pas nécessairement réentrantes face à une interruption asynchrone due à un signal, et que leur emploi dans un gestionnaire de signaux conduit à un comportement indéfini.

Pour résoudre ce problème, il est plus simple d'éviter d'utiliser un gestionnaire de signaux et de créer un thread spécifiquement chargé de la réception de tous les signaux – ou du moins d'une partie d'entre eux. Ce thread fonctionnera en boucle sur la fonction **sigwait()**.

```
int sigwait(const sigset_t * masque, int * numero_signal);
```

Cette routine fonctionne de façon identique à `sigwaitinfo()`, que nous avons rencontrée dans le chapitre 8, en attendant l'un des signaux contenus dans le masque passé en premier argument. Si un signal arrive, `sigwait()` se termine après avoir stocké le numéro du signal dans le pointeur passé en second argument. La fonction `sigwait()` étant un point d'annulation, on pourra laisser sans crainte un thread boucler dessus.

L'utilisation de `sigwait()` permet d'éviter l'emploi d'un gestionnaire. L'exécution du thread se poursuivant de manière normale, il est possible d'utiliser toutes les routines de la bibliothèque Pthreads dans une construction `switch/case`. Pour garantir un bon fonctionnement de `sigwait()`, il est indispensable que tous les autres threads bloquent les signaux attendus, évitant ainsi toute ambiguïté concernant le récepteur.

Conclusion

Dans ce chapitre nous avons essayé d'introduire les notions essentielles de la programmation multithread et de présenter les fonctions mises à disposition par la bibliothèque NPTL.

Pour une étude plus poussée concernant la norme Posix.1c, les circonstances de blocages, ou la conception même des programmes multithreads, on pourra consulter [NICHOLS 1996] *Pthreads Programming* ou [BUTENHOF 1997] *Programming with POSIX Threads*.

Des informations intéressantes peuvent également être trouvées dans les nombreuses FAQ Internet concernant les threads.

13

Gestion de la mémoire du processus

Nous allons nous intéresser dans ce chapitre à toutes les techniques permettant de gérer avec plus ou moins de précision l'espace mémoire d'un processus.

Nous commencerons par les principes d'allocation de mémoire dynamique. Ces mécanismes sont relativement classiques, peu différents des autres systèmes d'exploitation en ce qui concerne le programmeur applicatif. Par contre, la bibliothèque GlibC offre des possibilités puissantes pour le débogage, en assurant un suivi de toutes les allocations ou en permettant d'insérer notre propre code de surveillance dans le corps même des routines de gestion de la mémoire.

Des fonctionnalités avancées de manipulation de la mémoire seront examinées dans le chapitre suivant.

Indiquons rapidement que la gestion de la mémoire partagée, sujet connexe à celui de ce chapitre, sera étudiée ultérieurement avec les mécanismes de communication entre processus.

Routines classiques d'allocation et de libération de mémoire

Les variables utilisées dans un programme C peuvent être allouées de diverses manières :

- Les variables globales ou les variables déclarées statiques au sein des fonctions sont allouées une fois pour toutes lors du chargement du programme. Il existe même une différence entre les variables qui sont initialisées automatiquement au démarrage et celles qui n'ont pas de valeur initiale précise.
- Les variables locales et les arguments des fonctions voient leurs emplacements réservés dans la pile lors de l'invocation de la fonction.
- Les variables dynamiques sont allouées explicitement par l'intermédiaire des routines que nous allons étudier, à travers des pointeurs sur les zones réservées.

Le fait d'employer des variables dynamiques complique quelque peu la programmation, puisqu'il faut les manipuler au travers de pointeurs. De plus, elles doivent être allouées manuellement avant toute utilisation. Pourtant, il est nécessaire d'utiliser ces variables dans plusieurs cas :

- Lorsqu'on ne connaît pas la taille des variables lors de la compilation (par exemple une table contenant un nombre fluctuant d'éléments).
- Lorsqu'on a besoin d'allouer une zone mémoire de taille importante, principalement s'il s'agit d'une variable locale dans une fonction susceptible d'être invoquée de manière récursive, risquant un débordement de pile si on l'alloue de manière automatique.
- Lorsqu'on désire gérer la mémoire le plus finement possible, en réallouant les zones de mémoire au fur et à mesure des besoins, ou en utilisant des organisations des données telles que la liste chaînée, l'arbre binaire, la table de hachage…

Les méthodes d'allocation dynamique de mémoire n'ont rien de compliqué, mais il convient toutefois de prendre des précautions pour éviter les bogues de fuite mémoire, qui sont surtout sensibles avec des processus fonctionnant pendant de longues, voire de très longues durées.

Utilisation de malloc()

Pour allouer une nouvelle zone de mémoire, on utilise généralement la fonction **`malloc()`**, dont le prototype est déclaré dans `<stdlib.h>` ainsi :

```
void * malloc (size_t taille);
```

L'argument transmis correspond à la taille, en octets, de la zone mémoire désirée. Le type `size_t` étant non signé, il n'y a pas de risque de transmettre une valeur négative. Si on demande une taille valant zéro, la version Gnu de `malloc()` renvoie un pointeur `NULL`. Sinon, le système nous accorde une zone de la taille voulue et renvoie un pointeur sur cette zone. Si la mémoire disponible ne permet pas de faire l'allocation, `malloc()` renvoie un pointeur `NULL`. Il est fortement recommandé de tester le retour de toutes les demandes d'allocation. Le code demandant d'allouer une nouvelle structure de type `ma_struct_t` serait donc :

```
ma_struct_t * ma_struct;

if ((ma_struct = malloc(sizeof(ma_struct_t))) == NULL) {
    fprintf(stderr, "Pas assez de mémoire pour la structure voulue \n");
    exit(EXIT_FAILURE);
}
```

Pendant longtemps, il était d'usage de convertir explicitement le pointeur renvoyé par `malloc()` à l'aide d'un *cast* dans le type de pointeur désiré. Cette habitude disparaît car elle ne présente pas d'utilité (`malloc()` renvoie un pointeur `void *` qui peut se convertir automatiquement en pointeur sur n'importe quel type de données). Elle peut également masquer une petite erreur : si nous oublions d'inclure `<stdlib.h>`, `malloc()` est implicitement déclaré comme renvoyant une valeur de type `int`. Si nous ne faisons pas de *cast* explicite, le compilateur nous affichera un avertissement – ce qui nous permet de corriger l'oubli de `<stdlib.h>`.

Le problème qui se pose souvent au programmeur est de savoir quoi faire en cas d'échec d'allocation mémoire. En effet, il est possible que la mémoire du système se libère, lors de la

terminaison d'un processus gourmand, et que l'allocation réussisse si on la tente de nouveau quelques instants plus tard, mais on peut estimer aussi que l'état de surcharge du système est tel qu'il est probablement inutilisable, et qu'il vaut mieux terminer l'application au plus vite pour redonner la main à l'utilisateur, qui devra éliminer les processus consommant trop de ressources. L'échec d'une allocation dynamique est généralement le signe d'une fuite mémoire dans l'un des processus en cours, et il est préférable dans tous les cas de le signaler à l'utilisateur.

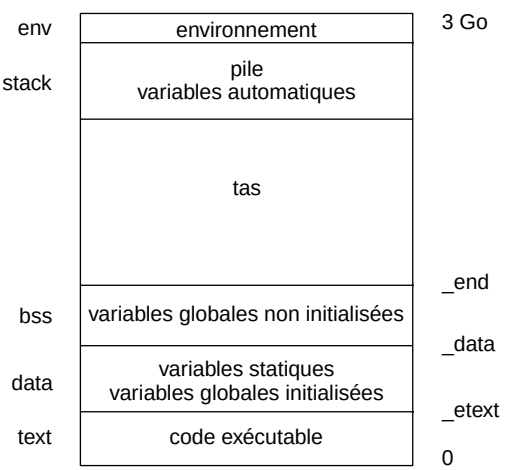
Malheureusement, dans certains cas extrêmes, le fait qu'une allocation mémoire ait réussi ne signifie pas que le processus puisse effectivement utiliser la mémoire qu'il croit disponible. Pour comprendre ce problème, il est nécessaire de s'intéresser au mécanisme détaillé de la gestion de la mémoire virtuelle sous Linux.

Un processus dispose d'un espace d'adressage linéaire immense, s'étendant jusqu'à 3 Go. Cet espace est découpé en segments ayant des rôles bien particuliers. Le processus peut connaître les limites de ses segments en utilisant des variables externes remplies par le chargeur de programmes du noyau :

- Le segment nommé **text** contient le code exécutable du processus ; il s'étend jusqu'à l'adresse contenue dans la variable `_etext`. Le début de ce segment varie selon le format de fichier exécutable. Dans le segment de code se trouvent également les routines des bibliothèques partagées utilisées par le processus.
- Le segment des données initialisées au chargement du processus et des données locales statiques des fonctions est nommé **data**. Il s'étend de l'adresse contenue dans la variable `_etext` jusqu'à celle contenue dans `_data`.
- Le segment des données non initialisées et des données allouées dynamiquement est nommé **bss**. Il s'étend de l'adresse contenue dans `_data` à celle contenue dans `_end`.

À l'autre bout de l'espace d'adressage se trouvent d'autres données comme les variables d'environnement et la pile du processus. Ces éléments ne nous concernent pas directement ici.

Figure 13.1
Espace d'adressage d'un processus



Lorsqu'on appelle la routine `malloc()`, de la bibliothèque C, celle-ci invoque l'appel-système **`brk()`**, qui est déclaré par le prototype suivant :

```
int brk (void * pointeur_end) ;
```

Cet appel-système permet de positionner la variable `_end`, modifiant ainsi la taille du segment *bss*. Il existe également une fonction de bibliothèque nommée **sbrk()** et déclarée ainsi :

```
void * sbrk (ptrdiff_t increment);
```

Celle-ci augmente ou diminue la taille du segment *bss* de l'incrément fourni en argument. En réalité, `malloc()` gère elle-même un ensemble de blocs mémoire qu'elle garde à disposition du processus, et ne fait appel à `sbrk()` qu'occasionnellement. Lorsque l'incrément est négatif, `sbrk()` sert à libérer de la mémoire. Le problème – si on peut dire – est que l'appel `sbrk()` n'échoue que très rarement. En effet, les cas d'erreur sont les suivants :

- Le processus essaye de libérer de la mémoire appartenant au segment de code *text*.
- Le processus essaye de dépasser sa limite `RLIMIT_DATA` fixée par la routine `setrlimit()` que nous avons vue dans le chapitre concernant l'exécution des programmes.
- La nouvelle zone de données va déborder sur une zone de projection mémoire telle que nous en verrons dans le prochain chapitre.
- L'allocation demandée excède la taille de la mémoire virtuelle globale du système (moins les valeurs minimales des buffers et du cache, ainsi qu'une marge de sécurité de 2 %).

Le problème principal se pose avec le dernier point. En effet, lorsque le noyau a augmenté la taille du segment de données d'un processus, il n'a pas pour autant réservé de la place effective dans la mémoire du système. Le principe de la mémoire virtuelle fait que c'est uniquement au moment où le processus tente d'écrire dans la zone nouvellement allouée que le système déclenche une faute d'accès et lui attribue une page mémoire réelle. Il est donc possible de réclamer beaucoup plus de mémoire que le système ne peut en fournir, sans pour autant que les allocations échouent.

> Ce comportement est dû à un mécanisme nommé *overcommit-memory* (sur-réservation mémoire) selon lequel on considère que les applications réclament toujours plus de mémoire qu'elles n'en ont effectivement besoin. On accepte alors toutes les allocations si l'espace d'adressage (mémoire virtuelle) du processus n'est pas saturé, et si la taille de la demande n'excède pas la somme de la mémoire physique disponible et du swap.
>
> Sans entrer dans les détails, indiquons que cela peut éviter des attaques de sécurité par déni de service. Lorsque toute la mémoire du système est saturée, un sous-ensemble du noyau nommé OOM-Killer (*Out Of Memory Killer*) est chargé de trouver le processus responsable de la saturation et de le tuer. Cela peut poser des problèmes de fiabilité dans des systèmes critiques, aussi est-il possible – depuis le noyau Linux 2.6 – de désactiver la sur-réservation mémoire. Pour cela il faut écrire dans le fichier `/proc/sys/vm/overcommit-memory` la valeur `2`. Pour plus de détails, on consultera le fichier `Documentation/vm/overcommit-acounting` dans les sources du noyau.

Si la machine dispose par exemple de 128 Mo de mémoire virtuelle, une demande d'allocation de 128 Mo en une fois échouera. Par contre, 128 demandes (ou plus) d'un Mo chacune seront acceptées tant qu'on n'aura pas essayé d'écrire réellement dans les zones allouées.

À titre d'exemple, ma machine actuelle contient 128 Mo de mémoire vive et 128 Mo de swap. Même sans prendre en compte le fait que le système fait tourner X-Window, Kde, un lecteur

de CD audio, la suite bureautique sur laquelle je rédige ce texte et plusieurs Xterms, je ne peux vraiment pas compter disposer dans une application de plus de 256 Mo de mémoire. Et pourtant le code suivant fonctionne sans erreur :

exemple_malloc_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    #define NB_BLOCS   257
    #define TAILLE    (1024*1024)

    int
main (void)
{
    int     i;
    char *  bloc [NB_BLOCS];

    for (i = 0; i < NB_BLOCS; i ++) {
        if ((bloc[i] = malloc(TAILLE)) == NULL) {
            fprintf(stderr, "Échec pour i = %d\n", i);
            break;
        }
    }
    fprintf(stderr, "Alloués : %d blocs de %d Ko \n", i, TAILLE / 1024);
    return EXIT_SUCCESS;
}
```

En voici l'exécution :

```
$ ./exemple_malloc_1
Alloués : 257 blocs de 1024 Ko
$
```

257 Mo alloués dans une mémoire virtuelle qui n'en comporte que 256, espace de swap compris !

Si nous essayons d'utiliser les zones allouées, par contre, le comportement est différent. Nous remplissons avec des zéros la mémoire renvoyée au fur et à mesure.

exemple_malloc_2. :

```
    #include <stdio.h>
    #include <stdlib.h>

    #define NB_BLOCS   257
    #define TAILLE    (1024*1024)

    int
main (void)
{
    int     i;
    char *  bloc [NB_BLOCS];

    for (i = 0; i < NB_BLOCS; i ++) {
```

```
        if ((bloc[i] = malloc(TAILLE)) == NULL) {
            fprintf(stderr, "Échec pour i = %d\n", i);
            break;
        }
        fprintf(stderr, "Remplissage de %d\n", i);
        memset(bloc[i], 0, TAILLE);
    }
    fprintf(stderr, "Alloués : %d blocs \n", i);
    return EXIT_SUCCESS;
}
```

Nous lançons le programme avec l'utilitaire `nice`, afin d'essayer de ne pas trop bloquer le reste du système (il faut quand même éviter de lancer ce processus sur une machine ouverte à plusieurs utilisateurs, le ralentissement du système dû à l'usage intensif du périphérique de swap est sensible).

```
$ nice ./exemple_malloc_2
Remplissage de 0
Remplissage de 1
Remplissage de 2
Remplissage de 3
Remplissage de 4
Remplissage de 5
  [...]
Remplissage de 198
Remplissage de 199
Remplissage de 200
Remplissage de 201
Remplissage de 202
Échec pour i = 203
Alloués : 203 blocs
$
```

Cette fois-ci, nous voyons que le programme échoue effectivement lorsqu'il n'y a plus de mémoire utilisable. Le problème c'est donc le risque qu'une allocation réussisse, alors qu'elle conduira par la suite à des dégradations sensibles des performances du système lorsqu'on tentera d'utiliser les zones allouées. Pour limiter au maximum cette éventualité, on essayera toujours de réclamer uniquement la mémoire dont on a réellement besoin au moment voulu, et on utilisera systématiquement les pages mémoire allouées le plus vite possible.

On pourrait être tenté d'utiliser par principe `calloc()` à la place de `malloc()`, car cette fonction effectue l'initialisation à zéro de toute la zone allouée. Pourtant, cela ne marcherait pas non plus dans certains cas. On notera tout d'abord que `calloc()` est une simple fonction de bibliothèque, et qu'il y a toujours un risque de voir un processus concurrent s'allouer une énorme zone de mémoire et la remplir aussitôt entre le moment où `calloc()` a fait appel à `sbrk()` et le moment où il initialise la nouvelle mémoire. Par ailleurs, pour des raisons d'efficacité, `calloc()` ne fait pas nécessairement des écritures en mémoire mais peut utiliser – surtout pour de grosses allocations – la fonction `mmap()`, que nous verrons dans le prochain chapitre, pour obtenir une zone remplie de zéros en projetant le périphérique `/dev/zero`.

La seule méthode vraiment efficace pour s'assurer la disponibilité des zones allouées est donc de toujours écrire rapidement dans les nouvelles données et d'éviter d'appeler successivement `malloc()` plusieurs fois de suite sans avoir rempli la mémoire fournie entre-temps.

On pourrait légitimement se demander si l'utilisation directe de `sbrk()` ne serait pas plus simple que celle de `malloc()`. En fait, la présentation que nous avons faite du rôle de `malloc()` – fonction de bibliothèque – vis-à-vis de l'appel-système `brk()` est largement simplificatrice. En fait, `malloc()` assure des fonctionnalités bien plus complexes que le simple agrandissement de la zone de données pour renvoyer un pointeur sur la mémoire allouée :

- Alignement : `malloc()` garantit que la mémoire fournie sera correctement positionnée afin de pouvoir y stocker n'importe quel type de donnée. Cela signifie que le processeur pourra manipuler directement les types entiers ou réels qu'on placera dans la mémoire allouée. Sur la plupart des machines, l'alignement des données est réalisé tous les 8 octets (taille d'un `double` ou d'un `long long int` sur les x86). Sur les architectures 64 bits, l'alignement est fixé tous les 16 octets.
- Configuration : `malloc()` offre de nombreuses possibilités de configuration de l'algorithme d'allocation, notamment en ce qui concerne le seuil où on passe d'une allocation avec `sbrk()` à une projection avec `mmap()`. De plus, `malloc()` permet à l'utilisateur de fournir ses propres points d'appel qui seront invoqués dans la routine. Cela permet d'inclure notre code de débogage personnalisé au cœur même des fonctions de bibliothèque.
- Vérification : la fonction `malloc()` et toutes les routines associées appliquent éventuellement leurs propres vérifications aux blocs alloués. Cela permet de s'assurer que l'application ne contient pas de fuites de mémoire.
- Optimisation : pour éviter le supplément de travail dû à l'appel-système `sbrk()`, la fonction `malloc()` réclame des blocs plus importants que nécessaire, afin de pouvoir en fournir directement une partie lors des invocations ultérieures.

De plus, `malloc()` utilise souvent l'appel-système `mmap()` pour obtenir de gros blocs de données indépendants, faciles à restituer au système lors de leur libération. Le fonctionnement de `mmap()` n'a rien à voir avec `brk()`.

Enfin, ajoutons que `malloc()` doit fonctionner correctement dans le cadre d'un processus déployant de multiples threads, en évitant les conflits d'accès simultanés à la limite de la zone de données. Pour toutes ces raisons, on voit que l'implémentation de la fonction `malloc()` est loin d'être triviale, et que les personnalisations éventuelles devront de préférence être apportées en utilisant les points d'entrée fournis par la bibliothèque GlibC plutôt qu'en tentant de réécrire une version bricolée de cette fonction.

Insistons sur un dernier point, avant de passer aux autres routines d'allocation mémoire, qui concerne l'utilisation de `malloc()` avec les chaînes de caractères. La bibliothèque C terminant toujours ses chaînes de caractères par un caractère nul, il est nécessaire d'allouer un octet de plus pour la nouvelle chaîne que la longueur désirée. Voici un exemple de fonction renvoyant une copie fraîchement attribuée de la chaîne passée en argument. Il sera du ressort de la fonction appelante de libérer la mémoire occupée par la copie lorsqu'elle n'en aura plus besoin :

```
    char *
alloue_et_copie_chaine (char * ancienne)
{
    char * nouvelle = NULL;

    if (ancienne != NULL) {
        nouvelle = malloc(strlen(ancienne) + 1);
        if (nouvelle != NULL)
            strcpy(nouvelle, ancienne);
```

```
    }
    return nouvelle;
}
```

Utilisation de calloc()

Le prototype de **calloc()** est le suivant :

```
void * calloc (size_t nb_elements, size_t taille_element);
```

Cette fonction sert principalement à allouer des tableaux. On fournit en premier argument le nombre d'éléments à accorder, et en second la taille d'un élément. En voici un exemple extrêmement classique :

exemple_calloc_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int *
calcul_fibonacci (int nombre_de_valeurs)
{
    int * table = NULL;
    int   i;
    if ((table = calloc(nombre_de_valeurs, sizeof (int))) == NULL) {
        fprintf(stderr, "Pas assez de mémoire \n");
        exit(EXIT_FAILURE);
    }
    if (nombre_de_valeurs > 0) {
        table[0] = 1;
        if (nombre_de_valeurs > 1) {
            table[1] = 1;
            for (i = 2; i < nombre_de_valeurs; i ++)
            table[i] = table[i - 2] + table[i - 1];
        }
    }
    return table;
}

    int
main (int argc, char * argv[])
{
    int   nb_valeurs;
    int * table;
    int   i;

    if ((argc != 2) || (sscanf(argv[1], "%d", & nb_valeurs) != 1)) {
        fprintf(stderr, "Syntaxe : %s nombre_de_valeurs\n", argv [0]);
        exit(EXIT_FAILURE);
    }
    table = calcul_fibonacci(nb_valeurs);
    for (i = 0; i < nb_valeurs; i ++)
        fprintf(stdout, "%d\n", table[i]);
```

```
    free(table);
    return EXIT_SUCCESS;
}
```

L'exécution affiche, on s'en doutait, le nombre désiré de termes de la suite de Fibonacci.

```
$ ./exemple_calloc_1 10
1
1
2
3
5
8
13
21
34
55
$
```

La fonction `calloc()` assure aussi, comme nous l'avons évoqué, que les zones allouées sont initialisées avec des zéros. Nous n'avons aucune garantie de ce genre avec les autres fonctions d'allocation. Elle est donc parfois préférée à `malloc()` pour s'affranchir des problèmes d'initialisation de variables, principalement lorsqu'on alloue dynamiquement des structures définies dans les fichiers d'en-tête d'autres modules, et qui sont susceptibles de posséder plus de membres que ceux qui sont utilisés par l'application. L'appel de `calloc()` permet ainsi d'initialiser toute la zone mémoire, y compris les membres dont nous n'avons pas nécessairement connaissance.

Nous l'avons déjà précisé, `calloc()` garantit que les zones allouées seront remplies avec des zéros, mais elle n'assure pas que cette initialisation se fera en utilisant des écritures effectives dans les zones mémoire reçues. Lorsque la taille de la zone concédée est suffisamment importante, `calloc()` utilise l'appel-système `mmap()` depuis le périphérique `/dev/zero`. Les pages allouées restent alors aussi virtuelles qu'avec `malloc()` jusqu'à ce qu'on écrive effectivement dedans. Pour plus de détails sur ce mécanisme, on se reportera au chapitre suivant. Voici un exemple qui reprend le principe de `exemple_malloc_1.c` :

exemple_calloc_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #define NB_BLOCS        257
    #define TAILLE          (1024*1024)

    int
main (void)
{
    int     i;
    char *  bloc[NB_BLOCS];

    for (i = 0; i < NB_BLOCS; i ++) {
        if ((bloc[i] = calloc(1, TAILLE)) == NULL) {
            fprintf(stderr, "Échec pour i = %d\n", i);
```

```
            break;
        }
    }
    fprintf(stderr, "Alloués : %d blocs de %d Ko \n", i, TAILLE / 1024);
    return EXIT_SUCCESS;
}
```

Le système nous alloue toujours aussi imperturbablement 257 Mo dans une mémoire virtuelle ne représentant que 256 Mo :

```
$ ./exemple_calloc_2
Alloués : 257 blocs de 1024 Ko
$
```

La fonction `calloc()` n'avait donc pas réellement touché aux zones allouées.

Si, par contre, nous échangeons les valeurs de `NB_BLOCS` et `TAILLE` afin d'allouer beaucoup de petites zones, `calloc()` utilise `sbrk()` suivi de `memset()`, avec donc une écriture effective sur les pages allouées :

```
$ nice ./exemple_calloc_3
Échec pour i = 803352
Alloués : 803352 blocs de 0 Ko
$
```

L'allocation échoue donc au bout d'un certain temps (196 Mo pour être précis).

Nous avons beaucoup insisté sur ces détails d'implémentation concernant l'écriture ou non dans les pages allouées, mais il est important lors d'une phase de débogage de comprendre les phénomènes sous-jacents si les allocations mémoire ont un comportement étrange.

Utilisation de realloc()

Il est souvent nécessaire de modifier en cours de fonctionnement la taille d'une table allouée dynamiquement. Pour cela, la bibliothèque C propose la fonction **`realloc()`**, très polyvalente, permettant de redimensionner aisément une zone de mémoire dynamique. Son prototype est le suivant :

```
void  * realloc (void * ancien, size_t taille);
```

Cette fonction crée une nouvelle zone de la taille indiquée et y recopie le contenu de l'ancienne zone. Elle renvoie ensuite un pointeur sur la nouvelle zone mémoire. Si la taille réclamée est supérieure à celle de l'ancien bloc, celui-ci est étendu, son contenu original se retrouvant au début de la nouvelle zone. Si l'allocation échoue, `realloc()` renvoie `NULL` mais ne touche pas à l'ancien bloc. Si, au contraire, l'allocation mémoire réussit, l'ancien pointeur n'est plus utilisable. Il faut donc employer une variable de stockage temporaire :

```
    void * nouveau;

    nouveau = realloc(bloc_de_donnees, nouvelle_taille);
    if (nouveau != NULL)
        bloc_de_donnees = nouveau;
    else
        fprintf(stderr, "Pas assez de mémoire \n") ;
```

Si la taille est inférieure à celle de l'ancien bloc, il y a libération de mémoire et l'ancienne zone de données est tronquée. Normalement une réduction ne doit pas échouer, mais cela peut quand même se produire sur certains systèmes où le nouveau bloc est alloué indépendamment de l'ancien avant d'y faire une copie. C'est surtout le cas quand des éléments de débogage importants sont ajoutés aux zones allouées.

Si la taille demandée est nulle, toute la mémoire est libérée et le pointeur renvoyé est `NULL`. Symétriquement, on peut transmettre un pointeur `NULL` en premier argument, et `realloc()` se conduit alors comme `malloc()`.

La fonction `realloc()` est particulièrement utile lorsque des données doivent être ajoutées ou supprimées au gré des actions de l'utilisateur. Imaginons un programme de dessin vectoriel où l'interface propose à l'utilisateur d'ajouter ou d'effacer des lignes. Une table stockée en variable globale et deux routines permettront de gérer proprement la mémoire :

```
    static ligne_t * table_lignes = NULL;
    static int       nb_lignes = 0;

    int
ajoute_ligne (void)
{    /*
      * Cette routine renvoie le numéro de la nouvelle ligne allouée
      * ou -1 en cas d'échec.
      */
    ligne_t * nouvelle;
    nouvelle = realloc(table_lignes, (nb_lignes + 1) * sizeof(ligne_t));
    if (nouvelle == NULL)
        return -1;
    table_lignes = nouvelle;
    nb_lignes ++;
    return nb_lignes - 1;
}

    void
supprime_ligne (int numero)
{
    ligne_t * nouvelle;
    if ((numero < 0) || (numero >= nb_lignes))
        return;
    if (numero != nb_lignes - 1)
        /*
         * Si on supprime un élément autre que le dernier,
         * on va recopier celui-ci dans l'ancien emplacement.
         */
        memcpy (& (table_lignes[numero]),
                & (table_lignes[nb_lignes -1]),
                sizeof(ligne_t));
    nouvelle = realloc(table_lignes, (nb_lignes - 1) * sizeof(ligne_t));
    if ((nouvelle != NULL) || (nb_lignes - 1 == 0))
        table_lignes= nouvelle;
    nb_lignes --;
}
```

Utilisation de free()

La plupart du temps, il faut libérer la mémoire qu'on a allouée dynamiquement. Cette libération s'effectue en invoquant la routine **free()** dont le prototype est :

```
void free (void * pointeur);
```

On transmet à `free()` un pointeur sur une zone mémoire qui a nécessairement été attribuée avec `malloc()`, `calloc()` ou `realloc()`[1]. Si on passe un pointeur `NULL`, `free()` ne fait rien (mais ce n'est pas une erreur).

Une fois qu'une zone a été libérée, il ne faut sous aucun prétexte essayer d'y faire de nouveau référence. De même, il ne faut pas non plus tenter de libérer plusieurs fois de suite la même zone, même si la version Gnu peut assurer une certaine tolérance vis-à-vis de ce genre de bogue. Il faut donc se méfier de la libération naïve d'une liste chaînée ainsi :

```
for (ptr = debut; ptr != NULL; ptr = ptr->suivant)
    free(ptr);
```

C'est une erreur grave, car le troisième membre de `for` fait référence à la zone pointée par `ptr` alors même que celle-ci a déjà été libérée. Il est nécessaire en fait de passer par une variable intermédiaire :

```
for (ptr = debut; ptr != NULL; ptr = suite) {
    suite = ptr->suivant;
    free(ptr);
}
```

Règles de bonne conduite pour l'allocation et la libération de mémoire

Il y a certains cas où on peut légitimement se demander s'il faut vraiment libérer la mémoire allouée dynamiquement. Après tout lorsqu'un processus se termine, tout son espace mémoire se libère et retourne au système d'exploitation. Si une variable est allouée à une seule reprise pour toute la durée du programme, il n'est pas indispensable de la libérer explicitement. On limitera quand même ce genre de comportement uniquement à des variables globales qui sont initialisées au démarrage du programme – par exemple en fonction de la valeur d'un argument en ligne de commande, ou suivant la valeur d'une variable d'environnement. L'idéal serait de restreindre l'allocation des blocs mémoire qu'on ne libère pas à la fonction `main()`. En effet, si le programme doit être ultérieurement modifié pour être utilisé en boucle, on verra tout de suite le problème lors de la mise à jour.

En règle générale, toute variable allouée dynamiquement devra être libérée à un moment ou à un autre. Il est essentiel, pour éviter les fuites de mémoire, d'adopter une attitude très précautionneuse lorsque l'allocation et la libération n'ont pas lieu dans la même fonction, ce qui est souvent le cas. De même, il faudra être très prudent avec les allocations dynamiques des membres de structures, elles aussi allouées dynamiquement.

Il est important de prendre de bonnes habitudes dans ces conditions. Nous présentons ici un exemple de « règles » de comportement vis-à-vis de la mémoire dynamique, mais chacun est libre d'adopter ses propres standards, du moment qu'on reste cohérent tout au long de l'application.

---

1. Ou toute autre fonction de la bibliothèque C – par exemple `tempnam()` – qui appelle l'une de ces routines. Cette particularité est normalement bien indiquée dans leur documentation.

- À chaque déclaration d'un pointeur, on l'initialise avec `NULL`. Ceci concerne également les membres des structures allouées dynamiquement s'il s'agit de pointeurs.
- Avant d'invoquer `malloc()`, on vérifie, éventuellement dans une condition `assert()`, que le pointeur à allouer est bien `NULL`.
- Après tout appel de `malloc()`, on s'assure qu'aucune erreur n'a eu lieu, sinon on gère le problème.
- Avant de libérer un pointeur, on vérifie – également dans un `assert()` - que le pointeur n'est pas `NULL`.
- Dès qu'on a libéré un pointeur avec `free()`, on le recharge immédiatement avec la valeur `NULL`.

Bien entendu, `malloc()` doit être considérée ici comme une fonction générique, et on adoptera la même attitude avec `realloc()` ou `calloc()`. Voici par exemple le genre de code qu'on peut produire :

```
    typedef struct element {
        char *          nom;
        struct element * suivant;
    } element_t;

    element_t * table = NULL;

    void
insere_element (char * nom)
{
    element_t * nouveau = NULL;
    /*
     * Si on insère du code ici, entre l'initialisation
     * du pointeur et l'allocation de la variable, il est
     * bon d'effectuer la vérification suivante.
     */
    assert(nouveau == NULL);
    nouveau = malloc(sizeof(element_t));
    if (nouveau == NULL) {
        /* traitement d'erreur */
        return;
    }
    nouveau->nom = NULL;
    nouveau->suivant = NULL;

    /* ... ici peut se trouver l'allocation de plusieurs membres ... */

    if (nom != NULL)  {
        nouveau->nom = malloc(strlen(nom) + 1);
        if (nouveau->nom != NULL)
            strcpy(nouveau->nom, nom);
        else {
            /* traitement d'erreur */
            free(nouveau);
            return;
        }
```

```
    }
    nouveau->suivant = table;
    table = nouveau;
}

    void
supprime_element (char * nom)
{
    element_t * elem = NULL;
    element_t * prec = NULL;

    if (nom == NULL)
        return;
    if (table == NULL)
        return;
    for (elem = table; elem != NULL; elem = elem->suivant) {
        if (strcmp(elem->nom, nom) == 0)
            break;
        prec = elem;
    }
    if (elem == NULL)
        /* pas trouvé */
        return;
    if (prec == NULL)
        /* pas de précédent : premier de la table */
        table = elem->suivant;
    else
        prec->suivant = elem->suivant;

    assert(elem->nom != NULL);
    free(elem->nom);
    elem->nom = NULL;
    /* ... Éventuellement libération d'autres membres ... */
    free(elem);
    elem = NULL;
}
```

Bien sûr, c'est de la programmation paranoïaque et maniaque, mais c'est souvent ce genre de routines qui se révèlent les plus robustes à l'usage, même si on perd largement en élégance de codage. En ce qui concerne les performances du logiciel, on remarquera que tous les tests peuvent être inclus uniquement dans la version de débogage, comme c'est le cas ici avec l'utilisation de `assert()`. De plus, on peut encadrer les initialisations par des directives conditionnelles `#ifndef NDEBUG` et `#endif`. La version de production du logiciel n'est donc pas pénalisée par des vérifications compulsives et redondantes.

Désallocation automatique avec alloca()

Il existe une alternative à l'utilisation du couple `malloc()-free()`, constituée par la fonction **`alloca()`**. Celle-ci présente le même prototype que `malloc()` :

```
void * alloca (size_t taille);
```

Le fonctionnement est identique à celui de `malloc()`, mais les zones de mémoire ne sont plus allouées dans le segment de données, mais à l'opposé dans la pile du processus. Rappelons que la mémoire du processus est constituée en bas du segment de code (*text*) et des segments de données initialisées (*data*) et non initialisées (*bss*). L'allocation avec `malloc()` fait croître ce dernier segment vers les adresses les plus élevées. À l'autre extrémité de l'espace d'adressage se trouvent les variables d'environnement et les arguments de la ligne de commande, tout en haut des 3 Go virtuels réservés au processus. En dessous de cette zone se situe le segment de pile, qui croît vers le bas, vers les adresses les plus petites.

Les données allouées avec `alloca()` sont placées dans le segment de pile du processus. L'avantage principal est que les zones allouées sont automatiquement libérées lors de la sortie de la fonction ayant invoqué `alloca()`. Il n'est plus nécessaire d'appeler `free()`, le retour de la fonction replace le pointeur de pile au-dessus des variables dynamiques, qui ne sont plus accessibles. On comprend bien d'ailleurs qu'il ne *faut pas* invoquer `free()` sur le pointeur renvoyé par `alloca()`, les domaines de travail de ces deux fonctions étant totalement disjoints.

Il n'est pas question d'allouer dynamiquement avec `alloca()` des variables globales ou une zone de mémoire sur laquelle la fonction doit renvoyer un pointeur. Seules les variables utilisées dans la fonction ou dans des sous-routines qu'on invoque peuvent être allouées correctement avec `alloca()`. Il ne faut pas non plus appeler directement `alloca()` dans les arguments d'une fonction qu'on invoque, car la zone allouée se trouverait, au sein de la pile, mélangée avec les arguments. Ceci est interdit :

```
appel_fonction (i, alloca(struct element));
```

Par contre, on peut utiliser :

```
struct element * elem;
elem = alloca(struct element);
appel_fonction(i, elem);
```

Le problème principal que pose `alloca()` est la gestion d'erreur. En effet, le système allouant automatiquement les pages nécessaires pour la pile, la seule erreur susceptible de se produire est le manque soudain de mémoire disponible. Le programme se retrouve dans la même situation que s'il avait invoqué en boucle infinie une routine récursive. Le processus risque alors de dépasser sa limite de taille maximale de pile `RLIMIT_STACK` renvoyée par `getrlimit()`. Il y a peu de chances que la gestion d'erreur classique (retour non `NULL`) fonctionne. Au contraire, le programme va recevoir un signal `SIGSEGV` qui le tuera. Voici un exemple de ce carnage :

exemple_alloca.c :

```
    #include <stdio.h>

    void
fonction_recursive (int iteration)
{
    char * bloc;

    fprintf(stdout, "Iteration %d\n", iteration);
    fflush(stdout);
    if ((bloc = alloca(512 * 1024)) == NULL) {
        fprintf(stdout, "Échec \n");
        return;
    }
```

```
    fonction_recursive(iteration + 1);
}

    int
main (void)
{
    fonction_recursive(1);
    return EXIT_SUCCESS;
}
```

Avant d'appeler la fonction, nous invoquons la commande `ulimit -s` du shell, qui nous donne la limite de taille de pile, en kilo-octets.

```
$ ulimit -s
8192
$ ./exemple_alloca
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11
Iteration 12
Iteration 13
Iteration 14
Iteration 15
Iteration 16
Segmentation fault (core dumped)
$
```

La limite étant de 8 192 Ko, soit 8 Mo, il est logique que notre programme ne puisse allouer correctement son seizième bloc de 512 Ko. Toutefois, on aurait préféré que `alloca()` nous renvoie simplement une valeur d'échec, plutôt que de voir le processus arrêté par un signal.

Un autre gros avantage de `alloca()` est de permettre la libération automatique même lorsqu'il y a un saut non local depuis une sous-routine. Imaginons un interpréteur de commandes. La routine principale est celle où on revient en cas de problème de syntaxe. Lorsqu'on décompose les commandes saisies pour les analyser, on fait appel à des sous-routines d'analyse lexicale. Si l'une de ces sous-routines découvre une erreur (mauvaise utilisation d'un mot clé réservé), elle peut déclencher un saut non local `siglongjmp()` pour revenir directement au plus haut niveau de l'interpréteur. Un problème se poserait alors pour les routines intermédiaires si elles ont alloué des données avec `malloc()`. Elles ne sont pas rappelées car l'analyseur lexical ne revient pas, et les données allouées ne sont pas libérées. Il est alors pratique d'utiliser des allocations avec libération automatique `alloca()`. Voici un exemple très simplifié :

```
    void
interpreteur (void)
{
```

```
    int erreur;
    while (1) {
        erreur = sigsetjmp(environnement_saut, 1);
        if (erreur != 0) {
            /* afficher un message d'erreur */
        }
        /* Saisie d'une commande */
        ...
        /* Appel de l'analyseur syntaxique */
        ...
        /* Exécution des commandes */
    }
}

    void
analyse_syntaxique (char * chaine);
{
    commande_t * table = NULL;
    commande_t * nouvelle = NULL;
    int cmd;

    /* construit une liste des commandes rencontrées */
    while (1) {
        /* si fin de chaine : retour */
        ...
        /* appel de l'analyseur lexical */
        ...
        nouvelle = alloca(sizeof(commande_t));
        nouvelle->suivante = table;
        table = nouvelle;
    }
}

    void
analyse_lexicale (char * chaine)
{
    /*  extraction des mots */
    ...
    /* si erreur d'entrée sortie -> retour à la boucle principale */
    if (erreur) {
        siglongjmp(environnement_saut, 1);
    }
    /* reste du traitement */
    ...
}
```

Le fait d'utiliser `alloca()` au lieu de `malloc()` permet dans ce cas une libération de la liste des commandes, car le saut non local restitue le pointeur de pile à la même position que durant l'invocation originale de `sigsetjmp()`. Même si on ne repasse pas par l'analyseur syntaxique, ses variables dynamiques sont libérées. Rappelons quand même que l'utilisation des sauts non locaux rend les programmes difficiles à lire et à déboguer, et qu'il vaut mieux les éviter au maximum.

Un dernier désagrément de `alloca()`est un léger manque de portabilité. Cette fonction est présente sur de nombreux systèmes Unix, mais elle n'est pas mentionnée dans les standards habituels.

Débogage des allocations mémoire

Dans un monde idéal, l'utilisation prudente de `malloc()` et de `free()` avec une vérification à chaque appel de l'état des pointeurs devrait suffire à éviter tout bogue de fuite de mémoire. Malheureusement, il en est rarement ainsi, et il existe toujours un risque d'erreur dans un programme où les variables sont allouées dynamiquement dans un module pour être libérées dans un autre module. C'est le cas, par exemple, pour toutes les routines utilitaires qui renvoient un pointeur sur un bloc mémoire fraîchement alloué, contenant les données désirées. Nous avons créé des fonctions de ce genre dans le chapitre sur les entrées-sorties, en guise de frontaux pour `sprintf()` et `fgets()`. À chaque utilisation de ces routines, il faut penser à libérer la mémoire renvoyée. Pour entretenir la confusion, il y a d'autres routines qui renvoient un pointeur sur des données statiques, à ne surtout pas libérer.

Même si le programme semble se comporter parfaitement, on aimerait quand même avoir la certitude que la mémoire est correctement gérée. L'observation « externe » du processus est malheureusement insuffisante, comme nous allons le vérifier. Nous allons créer un petit programme qui prend en argument deux valeurs et crée un tableau ayant le nombre d'éléments mentionnés en premier argument, chaque élément ayant la taille fournie en second argument. Ce programme invoque la commande `ps` pour afficher son propre état avant et après allocation. Ensuite, il libère tous les éléments, sauf le dernier alloué, et invoque `ps`. Puis il libère le dernier élément et affiche une dernière fois le résultat de `ps`. Nous analyserons ensuite son comportement suivant les diverses valeurs passées en argument. Bien sûr, nous remplirons les blocs alloués pour nous assurer qu'ils sont bien attribués physiquement au processus.

exemple_memoire.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (int argc, char * argv[])
{
    char    ligne_ps[80];
    char ** table = NULL;
    int     i;
    int     nb_blocs;
    int     taille_bloc;

    if ((argc != 3)
     || (sscanf(argv[1], "%d", & nb_blocs) != 1)
     || (sscanf(argv[2], "%d", & taille_bloc) != 1)) {
            fprintf(stderr, "Syntaxe : %s Nb_blocs Taille \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((nb_blocs < 1) || (taille_bloc < 1)) {
        fprintf(stderr, "Valeurs invalides \n");
```

```
        exit(EXIT_FAILURE);
}
    sprintf(ligne_ps, "ps un %ld", (long) getpid());
    fprintf(stdout, "Je démarre...\n");
    system(ligne_ps);

    fprintf(stdout, "J'alloue %d blocs de %d octets...",
             nb_blocs, taille_bloc);
    fflush(stdout);

    table = calloc(nb_blocs, sizeof(char *));
    if (table == NULL) {
        fprintf(stderr, "Échec \n");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < nb_blocs; i ++) {
        table[i] = malloc(taille_bloc);
        if (table[i] == NULL) {
            fprintf(stdout, "Échec \n");
            exit(EXIT_FAILURE);
        }
        memset(table[i], 1, taille_bloc);
    }
    fprintf(stdout, "Ok\n");
    system(ligne_ps);

    fprintf(stdout, "Je libère tous les blocs sauf le dernier \n");
    for (i = 0; i < nb_blocs - 1; i ++)
        free(table[i]);
    system(ligne_ps);

    fprintf(stdout, "Je libère le dernier bloc.. \n");
    free(table[nb_blocs - 1]);
    system(ligne_ps);
    return EXIT_SUCCESS;
}
```

Nous allons faire deux expériences : tout d'abord, nous essaierons deux allocations avec un petit nombre de gros blocs, puis nous réclamerons de nombreux petits blocs. Les champs qui nous intéressent dans la commande `ps` sont `VSZ` et `RSS`, qui représentent respectivement la taille totale de mémoire virtuelle utilisée par le processus et la place occupée en mémoire physique.

```
$ ./exemple_memoire
Syntaxe : ./exemple_memoire Nb_blocs Taille_bloc

$ ./exemple_memoire 100 1048576
Je démarre...
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   657  0.0  0.2  1052  376    S    13:21   0:00 ./exemple_memoire
J'alloue 100 blocs de 1048576 octets...Ok
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
```

```
 500   657 24.0 74.2 103852 95100  S    13:21   0:02 ./exemple_memoire
Je libère tous les blocs sauf le dernier
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   657 24.2  0.9  2080 1204    S    13:21   0:02 ./exemple_memoire
Je libère le dernier bloc..
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   657 24.2  0.1  1052  176    S    13:21   0:02 ./exemple_memoire

$ ./exemple_memoire 100 1048576
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   662  0.0  0.2  1052  376    S    13:21   0:00 ./exemple_memoire
J'alloue 100 blocs de 1048576 octets...Ok
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   662 53.5 80.6 103852 103204 S    13:21   0:01 ./exemple_memoire
Je libère tous les blocs sauf le dernier
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   662 36.0  1.1  2080 1432    S    13:21   0:01 ./exemple_memoire
Je libère le dernier bloc..
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   662 36.0  0.3  1052  404    S    13:21   0:01 ./exemple_memoire

$ ./exemple_memoire 102400 1024
Je démarre...
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   667  0.0  0.2  1052  376    S    13:22   0:00 ./exemple_memoire
J'alloue 102400 blocs de 1024 octets...Ok
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   667 31.2 81.2 104656 104008 S    13:22   0:01 ./exemple_memoire
Je libère tous les blocs sauf le dernier
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   667 26.6 81.2 104656 104008 S    13:22   0:01 ./exemple_memoire
Je libère le dernier bloc..
USER   PID %CPU %MEM   VSZ  RSS    STAT START   TIME COMMAND
 500   667 27.0  0.6  1456  808    S    13:22   0:01 ./exemple_memoire

$
```

Nous voyons que durant les deux premières invocations, nous réclamons 100 blocs d'un méga-octet chacun (1 048 576 octets). Pourtant, les deux invocations successives ne conduisent pas à la même occupation mémoire physique. Le système avait profité de la première invocation pour swapper des processus inutilisés (dont le traitement de texte que j'utilise pour écrire ces lignes !), et il disposait alors de plus de place dès le démarrage de la seconde invocation.

Le fait de ne pas libérer le dernier bloc n'a pas de répercussions sur les libérations précédentes. La taille des blocs (1 Mo) étant plus grande que la limite `M_MAP_THRESHOLD` (128 Ko) que nous rencontrerons dans le prochain paragraphe, l'algorithme de `malloc()` utilise des projections en mémoire avec `mmap()`, indépendantes les unes des autres.

Lors de la troisième invocation, nous allouons 100 K-blocs d'un kilo-octet chacun. La taille mémoire est donc identique à celle des deux premières expériences. Pourtant, la taille de l'espace utilisé, tant en mémoire virtuelle que physique, est modifiée. C'est dû à l'algorithme de `malloc()`. Les blocs étant plus petits qu'une page mémoire, il les attribue par groupes, en

utilisant `sbrk()`. Les blocs sont donc « empilés » les uns au-dessus des autres, le dernier se trouvant au sommet. Lorsque nous libérons toute la mémoire, sauf le dernier bloc, `malloc()` ne peut toujours pas faire redescendre la limite `_end` du segment de donnée, et nous voyons que la mémoire libérée n'est toujours pas revenue au système d'exploitation. Ce n'est qu'avec la dernière libération que la mémoire est restituée au noyau.

La conclusion de cette expérience est qu'il est difficile de vérifier depuis l'extérieur si un programme contient des fuites de mémoire. Il est nécessaire de disposer d'outils intégrés aux routines d'allocation pour surveiller le processus. C'est ce que nous verrons dans une prochaine section.

Configuration de l'algorithme utilisé par malloc()

La fonction **`mallopt()`** est déclarée ainsi dans `<malloc.h>` :

```
int mallopt (int parametre, int valeur);
```

Cette fonction permet de préciser une valeur pour un des paramètres utilisés par les routines d'allocation et de libération. Elle renvoie 1 si elle réussit, et 0 sinon. Les paramètres qu'on peut transmettre à `mallopt()` sont définis par les constantes symboliques suivantes :

Constante	Paramètre
`M_MMAP_MAX`	Le nombre maximal de blocs qui sont alloués en utilisant l'appel-système `mmap()` et non `sbrk()`. Ce paramètre peut être mis à zéro pour empêcher toute utilisation de `mmap()`. Sur certains systèmes, la capacité de projection avec `mmap()` peut être limitée. La valeur par défaut est de 1 024.
`M_MMAP_THRESHOLD`	Il s'agit de la taille de bloc à partir de laquelle on utilise `mmap()` et non plus `sbrk()`. L'avantage de l'emploi de `mmap()` est que la mémoire ainsi allouée retourne au système d'exploitation (nous l'avons observé) dès sa libération. L'inconvénient est que certains systèmes peuvent être limités en capacité de projection avec `mmap()`. Dans un contexte multithread, il est interdit de fixer le seuil à une valeur trop grande, car on risquerait de faire croître exagérément le segment de données, ce qui poserait des problèmes d'emplacement des multiples piles. Le seuil par défaut vaut 128 Ko.
`M_TOP_PAD`	Ce paramètre précise le volume mémoire supplémentaire que `malloc()` réclame au système lorsqu'elle appelle `sbrk()`. Cette mémoire supplémentaire sera donc disponible directement dans la fonction de bibliothèque lors des prochaines allocations sans avoir besoin d'invoquer l'appel-système. Par défaut, cette valeur est nulle.
`M_TRIM_THRESHOLD`	Il s'agit de la taille minimale d'un bloc à libérer pour qu'on appelle `sbrk()` avec une valeur négative. Pour éviter le surcoût d'un appel-système, `free()` ne libère effectivement la mémoire que lorsque le bloc est suffisamment conséquent. La valeur par défaut est de 128 Ko.

On peut obtenir exactement les mêmes effets en définissant, avant le premier appel à l'une des fonctions de la famille `malloc()`, les variables d'environnement suivantes (éventuellement depuis le shell) :

- `MALLOC_MMAP_MAX_`
- `MALLOC_MMAP_THRESHOLD_`
- `MALLOC_TOP_PAD_`
- `MALLOC_TRIM_THRESHOLD_`

Précisons quand même que la modification des paramètres de configuration de l'algorithme utilisé par `malloc()` est rarement nécessaire. Seuls des programmes effectuant de nombreuses allocations dynamiques dans des circonstances assez critiques peuvent avoir besoin de modifier ces données. Notons également qu'il n'est pas possible de lire les valeurs en cours. Les valeurs par défaut sont codées directement dans le fichier source `malloc.c` de la bibliothèque GlibC.

Suivi intégré des allocations et des libérations

Nous avons remarqué précédemment que l'observation externe des processus en cours d'exécution ne permettait pas de vérifier précisément si les allocations et libérations ne recelaient pas de fuites de mémoire. Les versions de `malloc()`, `calloc()`, `realloc()` et `free()` contenues dans la GlibC permettent d'enregistrer automatiquement toutes leurs actions dans un fichier externe. Ce fichier n'est pas conçu pour être lu directement par un utilisateur mais pour être analysé automatiquement par le script Perl `/usr/bin/mtrace` fourni avec la GlibC. Pour activer le suivi, il faut appeler la fonction **`mtrace()`**, dont le prototype est déclaré dans `<mcheck.h>` :

```
void mtrace (void);
```

Pour arrêter le suivi, on appelle **`muntrace()`** :

```
void muntrace(void);
```

Naturellement, on active souvent le suivi dès le début de la fonction `main()`, et on ne le désactive pas. Mais on peut ainsi retreindre le champ de l'analyse à une fonction particulière.

Lorsque `mtrace()` est appelée, elle recherche dans la variable d'environnement **`MALLOC_TRACE`** le nom d'un fichier sur lequel l'utilisateur a un droit d'écriture. Si le fichier existe, il est écrasé. Si `MALLOC_TRACE` n'est pas présente dans l'environnement du processus, ou si elle contient un nom de fichier invalide pour l'écriture, ou encore si le processus est installé avec les bits Set-UID ou Set-GID, `mtrace()` n'a pas d'effet.

Sinon, elle configure des routines spécifiques dans les points d'accès des routines `malloc()`, `realloc()` et `free()`, comme nous le verrons dans un prochain paragraphe. À chaque invocation de ces fonctions, des informations de débogage sont inscrites dans le fichier. À la fin du processus, on peut appeler l'utilitaire `mtrace` avec le nom de l'exécutable en argument, suivi du fichier de trace. Il présente alors les problèmes éventuels qui ont été détectés. Dans le programme de test suivant, nous introduisons une allocation à deux reprises dans le même pointeur (donc la première mémoire ne peut pas être libérée).

exemple_mtrace_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <mcheck.h>

    int
main (void)
{
    char * ptr;

    mtrace();
```

```
    if ((ptr = malloc(512)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    /* On écrase ptr, la première zone n'est plus libérable */
    if ((ptr = malloc(512)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    free(ptr);
    return EXIT_SUCCESS;
}
```

Et voici un exemple de session de débogage :

```
$ export MALLOC_TRACE="trace.out"
$ ./exemple_mtrace_1
$ mtrace exemple_mtrace_1 trace.out

Memory not freed:
-----------------
   Address     Size     Caller
0x08049750    0x200  at /home/ccb/src/ProgLinux/13/exemple_mtrace_1.c:13
```

La sortie de `mtrace` indique le fichier source fautif, ainsi que le numéro de ligne. Si on ne fournit pas le nom du fichier exécutable, `mtrace` affiche des résultats moins lisibles :

```
$ mtrace trace.out

Memory not freed:
-----------------
   Address     Size     Caller
0x08049750    0x200  at 0x80484b5
$
```

Lorsque le programme ne présente pas de défaut, `mtrace` l'indique.

exemple_mtrace_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <mcheck.h>

    int
main (void)
{
    char * ptr;

    mtrace();
    if ((ptr = malloc(512)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    free(ptr);
    return EXIT_SUCCESS;
}
```

Ici on nous indique qu'il n'y a pas de fuite de mémoire :

```
$ export MALLOC_TRACE=" trace.out"
$ ./exemple_mtrace_2
$ mtrace trace.out
No memory leaks.
$
```

Nous avons signalé que lorsque `mtrace()` est invoquée sans que la variable d'environnement `MALLOC_TRACE` ne contienne de nom de fichier correct, la fonction ne faisait rien. Ce n'est pas pour autant une raison pour appeler systématiquement `mtrace()` au début de nos programmes. En effet, il y aura un conflit le jour où on lancera, sans y penser, le programme durant une session de débogage d'une autre application. La variable d'environnement pointant vers le même fichier de suivi, les traces des deux processus seront inextricablement mélangées. La meilleure attitude à adopter est d'encadrer l'appel à `mtrace()`par des directives de compilation conditionnelles :

```
#ifndef NDEBUG
    mtrace();
#endif
```

Ainsi, lorsque la phase de débogage sera terminée, une recompilation permettra d'éliminer automatiquement l'appel à `mtrace()` du code de distribution. Il est important de remarquer qu'une fois que `mtrace()` a été appelée, l'ensemble des fonctions de bibliothèque qui invoquent `malloc()` sont également concernées par le suivi. Il peut s'agir bien entendu de la GlibC, mais également de n'importe quelle autre bibliothèque utilisée par le programme. Certaines de ces fonctions peuvent allouer de la mémoire qu'elles libéreront en une seule fois, à la fin du processus, avec une routine enregistrée par `atexit()`. Si on utilise `muntrace()` avant que la fonction `main()` ne se termine, on risque d'obtenir des messages d'alarme excessifs de `mtrace`. On a donc intérêt à éviter au maximum l'appel de `muntrace()`, sauf lorsqu'on désire déboguer une partie précise du programme.

Notons aussi que `mtrace()` indique toutes les variables dynamiques allouées mais qui n'ont pas été libérées explicitement. Or, il arrive fréquemment, tant dans les programmes applicatifs que dans les bibliothèques système, que des buffers soient alloués automatiquement au lancement du processus, mais qu'ils ne soient pas libérés avant que la fin du programme ne restitue toute la mémoire au système. On ne s'étonnera donc pas que ces allocations soient signalées à chaque fois. La surveillance des programmes utilisant par exemple les bibliothèques Xlib, Xt ou les fonctions réseau de la bibliothèque C est rendue un peu pénible. En règle générale, on ne s'occupera que des allocations de son propre programme, qu'on filtrera avec `/bin/grep`.

Surveillance automatique des zones allouées

La fonction **`mcheck()`** est déclarée dans `<mcheck.h>` ainsi :

```
int mcheck (void (* fonction_d_erreur) (enum mcheck_status status));
```

On passe à cette routine un pointeur sur une fonction qui sera automatiquement invoquée lorsque l'une des fonctions de la famille `malloc()` détecte une incohérence dans un bloc de mémoire dynamique. Si on passe un pointeur `NULL`, `mcheck()` installe un gestionnaire d'erreur par défaut qui affiche simplement un message sur `stderr` avant d'invoquer `abort()`.

Comme la fonction `mcheck()` installe des routines de vérification dans les points d'entrée des fonctions d'allocation, elle doit être appelée avant toute utilisation de `malloc()`. Il faut donc la placer le plus tôt possible dans la fonction `main()`. Cela peut poser un problème en C++, où les constructeurs d'objets statiques peuvent appeler `malloc()` avant l'entrée dans la fonction `main()`. On peut alors utiliser l'option `-lmcheck` au moment de l'édition des liens du programme, ce qui correspond à un `mcheck(NULL)` dès l'initialisation du processus.

Si `mcheck()` réussit, elle renvoie 0, si elle échoue parce qu'on l'a appelée trop tard – autrement dit après un premier `malloc()` – elle renvoie –1. Les vérifications ont lieu automatiquement lorsqu'on invoque une des fonctions de la famille `malloc()`, mais on peut également demander un contrôle immédiat d'un bloc mémoire en utilisant **`mprobe()`** déclarée ainsi :

```
void mprobe(void * pointeur);
```

On lui transmet bien entendu un pointeur sur la zone à tester. Lorsqu'une erreur est détectée, notre routine est appelée avec en argument l'une des valeurs suivantes :

Valeur	Signification
`MCHECK_DISABLED`	`mcheck()` a été appelée trop tard, on ne peut plus faire de vérification. Cette valeur n'est transmise que sur une demande `mprobe()`.
`MCHECK_OK`	Pas d'erreur.
`MCHECK_HEAD`	On a détecté que le bloc de mémoire placé juste avant celui qu'on examine a été écrasé. C'est un cas courant de boucle dans laquelle le compteur est descendu par erreur jusqu'à -1 au lieu de s'arrêter à zéro.
`MCHECK_TAIL`	Inversement, le bloc de mémoire placé après celui qu'on surveille a été touché. Cela peut se produire par exemple lorsqu'on alloue dynamiquement la mémoire pour une chaîne en oubliant de compter le caractère nul final.
`MCHECK_FREE`	Le bloc examiné a déjà été libéré.

Voyons un exemple de surveillance automatique.

exemple_mcheck_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <mcheck.h>

    void fonction_d_erreur (enum mcheck_status status);

    #define NB_INT  20

    int
main (void)
{
    int * table_int;
    int     i;

    if (mcheck(fonction_d_erreur) != 0) {
        perror("mcheck");
        exit(EXIT_FAILURE);
    }
```

```
    fprintf(stdout, "Allocation de la table \n");
    table_int = calloc(NB_INT, sizeof(int));

    fprintf(stdout, "On déborde vers le haut \n");
    for (i = 0; i <= NB_INT; i ++)
        table_int[i] = 1;

    fprintf(stdout, "Libération de la table \n");
    free(table_int);

    fprintf(stdout, "Allocation de la table \n");
    table_int = calloc(NB_INT, sizeof(int));

    fprintf(stdout, "On déborde vers le bas \n");
    i = NB_INT;
    while (i >= 0)
        table_int[--i] = 1;

    fprintf(stdout, "Libération de la table \n");
    free(table_int);

    fprintf(stdout, "Allocation de la table \n");
    table_int = calloc(NB_INT, sizeof(int));

    fprintf(stdout, "Écriture normale \n");
    for (i = 0; i < NB_INT; i ++)
        table_int[i] = 0;

    fprintf(stdout, "Libération de la table \n");
    free(table_int);

    fprintf(stdout, "Et re-libération de la table ! \n");
    free(table_int);
    return EXIT_SUCCESS;
}

    void
fonction_d_erreur (enum mcheck_status status)
{
    switch(status) {
        case MCHECK_DISABLED :
            fprintf(stdout, " -> Pas de vérification \n");
            break;
        case MCHECK_OK :
            fprintf(stdout, " -> Vérification Ok \n");
            break;
        case MCHECK_HEAD :
            fprintf(stdout, " -> Données avant un bloc écrasées \n");
            break;
        case MCHECK_TAIL :
            fprintf(stdout, " -> Données après un bloc écrasées \n");
            break;
```

```
        case MCHECK_FREE :
            fprintf(stdout, " -> Bloc déjà libéré \n");
            break;
    }
}
```

Voici l'exécution de notre programme :

```
$ ./exemple_mcheck_1
Allocation de la table
On déborde vers le haut
Libération de la table
 -> Données après un bloc écrasées
Allocation de la table
On déborde vers le bas
Libération de la table
 -> Données avant un bloc écrasées
Allocation de la table
Écriture normale
Libération de la table
Et re-libération de la table !
 -> Bloc déjà libéré
Segmentation fault (core dumped)
$
```

Notons que l'arrêt brutal du programme est dû à la double libération du dernier pointeur. Cette erreur avait été détectée lors de l'appel de `free()`, mais comme nous n'avons rien fait d'autre que d'afficher un message, la seconde tentative de libération a eu lieu normalement.

Il est aussi possible de demander aux routines d'allocation d'effectuer le même genre de surveillance simplement en définissant la variable d'environnement **`MALLOC_CHECK_`**. Dès que cette variable est définie, les routines d'allocation deviennent plus tolérantes, permettant les multiples libérations d'un même bloc ou les débordements d'un octet en haut ou en bas d'un bloc. Ensuite, en fonction de la valeur de `MALLOC_CHECK_`, le comportement varie lorsqu'une erreur est rencontrée :

- Si `MALLOC_CHECK_` vaut 1, un message est inscrit sur la sortie d'erreur standard.
- Si `MALLOC_CHECK_` vaut 2, le message est inscrit, puis le processus est arrêté avec un `abort()`. Le fichier `core` créé permettra de retrouver l'endroit où l'erreur s'est produite.
- Pour toute autre valeur de `MALLOC_CHECK_`, l'erreur est silencieusement ignorée. Autant dire que ce n'est pas une méthode raisonnable pour éliminer un dysfonctionnement (quoique cela assure une certaine protection pendant une démo !).

Le programme `exemple_mcheck_2.c` effectue les mêmes opérations que `exemple_mcheck_1.c`, mais il ne met pas en place de gestionnaire d'erreur. De plus, la table allouée contient des caractères et non plus des entiers, ce qui permet de rentrer dans le cadre de tolérance de `MALLOC_CHECK_` pour les débordements d'un octet. Voici un exemple d'exécution de ce programme avec diverses configurations de la variable d'environnement :

```
$ unset MALLOC_CHECK_
$ ./exemple_mcheck_2
Allocation de la table
On déborde vers le haut
```

```
Libération de la table
Allocation de la table
On déborde vers le bas
Libération de la table
Segmentation fault (core dumped)
$ export MALLOC_CHECK_=1
$ ./exemple_mcheck_2
Allocation de la table
malloc: using debugging hooks
On déborde vers le haut
Libération de la table
free(): invalid pointer 0x8049830!
Allocation de la table
On déborde vers le bas
Libération de la table
free(): invalid pointer 0x8049850!
Allocation de la table
Écriture normale
Libération de la table
Et re-libération de la table !
free(): invalid pointer 0x8049870!
$ export MALLOC_CHECK_=2
$ ./exemple_mcheck_2
Allocation de la table
On déborde vers le haut
Libération de la table
Aborted (core dumped)
$ export MALLOC_CHECK_=0
$ ./exemple_mcheck_2
Allocation de la table
On déborde vers le haut
Libération de la table
Allocation de la table
On déborde vers le bas
Libération de la table
Allocation de la table
Écriture normale
Libération de la table
Et re-libération de la table !
$
```

La première exécution (`MALLOC_CHECK_` non définie) échoue lors de la tentative de double libération du pointeur. La seconde (valeur 1) affiche les erreurs, mais les tolère. La troisième exécution (valeur 2) affiche les erreurs et s'arrête dès qu'une incohérence est rencontrée. Enfin, la dernière exécution (valeur quelconque, 0 en l'occurrence) autorise toutes les erreurs sans afficher de message.

Fonctions d'encadrement personnalisées

Nous avons indiqué que les fonctionnalités de surveillance, comme `mcheck()`, utilisent des points d'entrée dans les routines d'allocation et de libération pour insérer leur code. Mais nous pouvons également utiliser ces points d'entrée pour y glisser nos propres fonctions de

supervision. Ce genre de procédé de débogage est relativement pointu et n'est généralement nécessaire que pour des logiciels vraiment conséquents, où un processus particulier est, par exemple, chargé de surveiller le déroulement de ses confrères. L'insertion d'une routine de débogage se fait en utilisant les variables globales suivantes, déclarées dans `<malloc.h>` :

Variable	Utilisation
`__malloc_hook`	Pointeur sur une fonction du type `void * fonction (size_t taille, void * appelant)` Cette routine sera appelée à la place de `malloc()` ; elle reçoit en argument la taille de bloc à allouer et un pointeur contenant l'adresse de retour, ce qui permet de retrouver l'emplacement de l'appel fautif si une erreur est détectée. Cette fonction doit renvoyer un pointeur sur la zone de mémoire nouvellement allouée. Nous montrerons plus bas comment invoquer l'ancienne routine d'allocation pour obtenir le bloc mémoire désiré.
`__realloc_hook`	Pointeur sur une routine de type `void *fonction(void *ancien,size_t taille, void *appel)` Cette fonction sera appelée lorsqu'on invoquera `realloc()` et devra s'occuper de redimensionner l'ancien bloc avec la nouvelle taille désirée. L'emplacement où on a fait appel à `realloc()` est transmis en troisième argument pour retrouver une éventuelle erreur.
`__free_hook`	Pointeur sur une routine de type `void fonction (void * pointeur, void * appelant)` chargée de libérer le bloc de mémoire correspondant au pointeur transmis.

Nous avons parlé de fonctions d'encadrement personnalisées, et non de fonctions d'allocation et de libération personnalisées. En effet, bien qu'il soit possible d'écrire nos propres routines de gestion complète de la mémoire, ce travail serait très difficile, et nous allons nous contenter d'insérer du code servant de tremplin pour l'appel des véritables routines `malloc()`, `realloc()` et `free()`. Dans notre exemple, nous nous contenterons d'afficher sur la sortie d'erreur standard les appels effectués. Les routines d'encadrement pourraient être bien plus subtiles, en vérifiant l'intégrité des blocs mémoire par exemple, comme nous le verrons plus loin.

Pour pouvoir faire appel aux routines originales `malloc()`, `realloc()` ou `free()`, il est nécessaire de stocker dans des variables globales les valeurs initiales des points d'entrée. Lorsque nous désirerons les invoquer, il suffira de restituer les valeurs originales des points d'entrée et de faire un appel normal à la fonction concernée. Comme nous ne connaissons pas les interdépendances entre les routines de la bibliothèque, il faudra à chaque fois sauver, modifier et restituer l'ensemble des trois points d'entrée.

On notera aussi qu'au retour d'une des véritables fonctions d'allocation, il nous faudra sauver à nouveau les trois points d'entrée et réinstaller nos routines. En effet, une routine comme `malloc()` peut pointer à l'origine sur une fonction d'initialisation qui, après son exécution, modifiera son propre point d'entrée pour accéder directement au code d'allocation durant les invocations ultérieures. Une routine peut donc modifier son propre point d'entrée ou celui des autres fonctions, et on sauvera de nouveau à chaque fois les trois pointeurs. Voyons donc un exemple de fonctions d'encadrement.

exemple_hook.c :

```
#include <stdio.h>
#include <stdlib.h>
```

```
    #include <malloc.h>

    static void * pointeur_malloc  = NULL;
    static void * pointeur_realloc = NULL;
    static void * pointeur_free    = NULL;

    static void * mon_malloc  (size_t taille, void * appel);
    static void * mon_realloc (void * ancien, size_t taille,
                               void appel);
    static void   mon_free (void * pointeur, void * appel);

    int
main (void)
{
    char * bloc;

    /* Installation originale */
#ifndef NDEBUG
    pointeur_malloc = __malloc_hook;
    pointeur_realloc = __realloc_hook;
    pointeur_free = __free_hook;
    __malloc_hook = mon_malloc;
    __realloc_hook = mon_realloc;
    __free_hook = mon_free;
#endif
    /* et maintenant quelques appels... */
    bloc = malloc(128);
    bloc = realloc(bloc, 256);
    bloc = realloc(bloc, 16);
    free(bloc);
    bloc = calloc(256, 4);
    free(bloc);
    return EXIT_SUCCESS;
}

    static void *
mon_malloc (size_t taille, void * appel)
{
    void * retour;

    /* restitution des pointeurs et appel de l'ancienne routine */
    __malloc_hook = pointeur_malloc;
    __realloc_hook = pointeur_realloc;
    __free_hook = pointeur_free;
    retour = malloc(taille);
    /* Écriture d'un message sur stderr */
    fprintf(stderr, "%p : malloc(%u) -> %p \n", appel, taille, retour);
    /* on réinstalle nos routines */
    pointeur_malloc = __malloc_hook;
    pointeur_realloc = __realloc_hook;
    pointeur_free = __free_hook;
```

```
    __malloc_hook = mon_malloc;
    __realloc_hook = mon_realloc;
    __free_hook = mon_free;
    return retour;
}

    static void *
mon_realloc (void * ancien, size_t taille, void * appel)
{
    void * retour;
    /* restitution des pointeurs et appel de l'ancienne routine */
    __malloc_hook = pointeur_malloc;
    __realloc_hook = pointeur_realloc;
    __free_hook = pointeur_free;
    retour = realloc(ancien, taille);
    /* Écriture d'un message sur stderr */
    fprintf(stderr, "%p : realloc(%p, %u) -> %p \n",
                appel, ancien, taille, retour);
    /* on réinstalle nos routines */
    pointeur_malloc = __malloc_hook;
    pointeur_realloc = __realloc_hook;
    pointeur_free = __free_hook;
    __malloc_hook = mon_malloc;
    __realloc_hook = mon_realloc;
    __free_hook = mon_free;
    return retour;
}

    static void
mon_free (void * pointeur, void * appel)
{
    /* restitution des pointeurs et appel de l'ancienne routine */
    __malloc_hook = pointeur_malloc;
    __realloc_hook = pointeur_realloc;
    __free_hook = pointeur_free;
    free(pointeur);
    /* Écriture d'un message sur stderr */
    fprintf(stderr, "%p : free(%p)\n", appel, pointeur);
    /* on réinstalle nos routines */
    pointeur_malloc = __malloc_hook;
    pointeur_realloc = __realloc_hook;
    pointeur_free = __free_hook;
    __malloc_hook = mon_malloc;
    __realloc_hook = mon_realloc;
    __free_hook = mon_free;
}
```

Le fait d'utiliser l'argument `void * appel` dans les routines est en fait une astuce permettant de récupérer l'adresse de retour directement dans la pile. En fait, les variables `__malloc_hook`, `__realloc_hook` et `__free_hook` sont conçues pour stocker des pointeurs sur des fonctions ne comportant pas ce dernier argument. Il ne faut donc pas s'étonner des avertissements fournis par le compilateur. On peut les ignorer sans danger.

On remarquera que l'encadrement par `#ifndef NDEBUG #endif` de l'initialisation de nos routines permet d'éliminer ce code de débogage lors de la compilation pour la version de distribution du logiciel. Voici à présent un exemple d'exécution :

```
$ make
cc -Wall -g    exemple_hook.c   -o exemple_hook
exemple_hook.c: In function `main':
exemple_hook.c:24: warning: assignment from incompatible pointer type
exemple_hook.c:25: warning: assignment from incompatible pointer type
exemple_hook.c:26: warning: assignment from incompatible pointer type
exemple_hook.c: In function `mon_malloc':
exemple_hook.c:56: warning: assignment from incompatible pointer type
exemple_hook.c:57: warning: assignment from incompatible pointer type
exemple_hook.c:58: warning: assignment from incompatible pointer type
exemple_hook.c: In function `mon_realloc':
exemple_hook.c:79: warning: assignment from incompatible pointer type
exemple_hook.c:80: warning: assignment from incompatible pointer type
exemple_hook.c:81: warning: assignment from incompatible pointer type
exemple_hook.c: In function `mon_free':
exemple_hook.c:99: warning: assignment from incompatible pointer type
exemple_hook.c:100: warning: assignment from incompatible pointer type
exemple_hook.c:101: warning: assignment from incompatible pointer type
$ ./exemple_hook
0x804859c : malloc(128) -> 0x8049948
0x80485b2 : realloc(0x8049948, 256) -> 0x8049948
0x80485c5 : realloc(0x8049948, 16) -> 0x8049948
0x80485d6 : free(0x8049948)
0x80485e5 : malloc(1024) -> 0x8049948
0x80485f6 : free(0x8049948)
$
```

Nous voyons bien tous nos appels aux trois routines de surveillance et aussi, que `calloc()` est construit en invoquant `malloc()`, ce qui est rassurant car il n'existe pas de point d'entrée `__calloc_hook`. Les adresses fournies lors de l'invocation peuvent paraître particulièrement obscures, mais on peut aisément utiliser `gdb` pour retrouver la position de l'appel dans le programme. Recherchons par exemple où se trouve le second `realloc()` :

```
$ gdb exemple_hook
GNU gdb 4.17.0.11 with Linux support
  [...]
(gdb) list *0x80485c5
0x80485c5 is in main (exemple_hook.c:32).
27  #endif
28
29          /* et maintenant quelques appels... */
30          bloc = malloc(128);
31          bloc = realloc(bloc, 256);
32          bloc = realloc(bloc, 16);
33          free(bloc);
34          bloc = calloc(256, 4);
35          free(bloc);
36
```

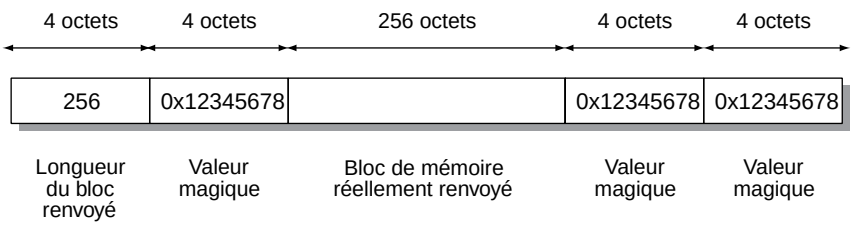
```
(gdb) quit
$
```

En entrant `list *` suivi de l'adresse recherchée, `gdb` nous indique bien qu'il s'agit de la ligne 32 du fichier `exemple_hook.c`, dans la fonction `main()`.

Mais que d'énergie déployée pour obtenir *grosso modo* le même résultat qu'en invoquant `mtrace()` en début de programme et en définissant la variable d'environnement `MALLOC_TRACE` ! En fait, nous pouvons utiliser ces points d'entrée dans les routines de gestion mémoire pour effectuer des vérifications d'intégrité beaucoup plus poussées. On peut être confronté à des débordements de buffer d'un seul octet, par exemple à cause d'une mauvaise borne supérieure d'un intervalle, d'une utilisation de l'opérateur `<=` au lieu de `<`, ou tout simplement en oubliant de compter le caractère nul à la fin d'une chaîne. Ces bogues sont difficiles à détecter car, du fait de l'alignement des blocs sur des adresses multiples de 8 ou de 16 octets suivant les architectures, le débordement d'un seul caractère peut parfaitement passer inaperçu pendant longtemps.

Nous allons considérer ici qu'un `long int` occupe 4 octets, ce qui est le cas sur les architectures x86, mais on peut reprendre le même raisonnement en utilisant simplement `sizeof(long int)` pour rester portable. Nous pouvons encadrer toutes les zones de mémoire allouées par deux blocs de 8 octets supplémentaires. Le bloc inférieur contiendra la longueur de la mémoire allouée (sur 4 octets) et une valeur constante sur les 4 octets suivants. Le bloc supérieur comprendra deux fois cette valeur constante. La zone de mémoire renvoyée à l'appelant sera comprise entre les deux blocs de surveillance. Imaginons que le programme demande une allocation de 256 octets, nous en avons en réalité 256+16, soit 272 octets organisés comme sur la figure 13.2.

Figure 13.2
Encadrement du bloc alloué



Lorsque nous allouerons un bloc, nous remplirons les données de surveillance correctement. Ensuite, quand nous recevrons un pointeur sur un bloc à redimensionner ou à libérer, nous vérifierons que les données sont toujours en place. Si ce n'est pas le cas, le processus est arrêté avec `abort()`. Avant de libérer un bloc, nous écraserons toutes les données qu'il contient pour nous assurer qu'il n'est plus valide. Le programme `exemple_hook_2.c` va faire déborder une copie de chaîne en oubliant volontairement le caractère nul.

exemple_hook_2.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>

static void * pointeur_malloc           = NULL;
```

```
    static void * pointeur_realloc  = NULL;
    static void * pointeur_free             = NULL;

    static void * mon_malloc  (size_t taille);
    static void * mon_realloc (void * ancien, size_t taille,
                               void * appel);
    static void   mon_free (void * pointeur, void * appel);
    static int    verifie_pointeur (void * pointeur);

    #define RESTITUTION_POINTEURS() __malloc_hook = pointeur_malloc;   \
                                    __realloc_hook = pointeur_realloc; \
                                    __free_hook = pointeur_free;
    #define SAUVEGARDE_POINTEURS()  pointeur_malloc = __malloc_hook;   \
                                    pointeur_realloc = __realloc_hook; \
                                    pointeur_free = __free_hook;
    #define INSTALLATION_ROUTINES() __malloc_hook = mon_malloc;   \
                                    __realloc_hook = mon_realloc; \
                                    __free_hook = mon_free
     int
main (void)
{
    char * bloc = NULL;
    char * chaine = "chaine à copier";

    /* Installation originale */
#ifndef NDEBUG
    SAUVEGARDE_POINTEURS();
    INSTALLATION_ROUTINES();
#endif
    /* Une copie avec oubli du caractère final */
    bloc = malloc(strlen (chaine));
    if (bloc != NULL)
        strcpy(bloc, chaine);
    free(bloc);
    return EXIT_SUCCESS;
}

    #define VALEUR_MAGIQUE  0x12345678L

    static void *
mon_malloc (size_t taille)
{
    void * bloc;

    RESTITUTION_POINTEURS();
    bloc = malloc(taille + 4 * sizeof(long));
    SAUVEGARDE_POINTEURS();
    INSTALLATION_ROUTINES ();
    if (bloc == NULL)
        return NULL;
    /* et remplissage des données supplémentaires */
    * (long *) bloc = taille;
```

```
    * (long *) (bloc  + sizeof(long)) = VALEUR_MAGIQUE;
    * (long *) (bloc + taille + 2 * sizeof(long)) = VALEUR_MAGIQUE;
    * (long *) (bloc + taille + 3 * sizeof(long)) = VALEUR_MAGIQUE;
    /* on renvoie un pointeur sur le bloc réservé à l'appelant */
    return (bloc + 2 * sizeof(long));
}

    static void *
mon_realloc (void * ancien, size_t taille, void * appel)
{
    void * bloc;

    if (! verifie_pointeur(ancien) {
        fprintf(stderr, "%p : realloc avec mauvais bloc\n", appel);
        abort();
    }
    RESTITUTION_POINTEURS();
    if (ancien != NULL)
        bloc = realloc(ancien - 2 * sizeof(long),
                       taille + 4 * sizeof(long));
    else
        bloc = malloc(taille + 4 * sizeof(long));
    SAUVEGARDE_POINTEURS();
    INSTALLATION_ROUTINES ();
    if (bloc == NULL)
        return bloc;
    /* et remplissage des données supplémentaires */
    * (long *) bloc = taille;
    * (long *) (bloc  + sizeof(long)) = VALEUR_MAGIQUE;
    * (long *) (bloc + taille + 2 * sizeof(long)) = VALEUR_MAGIQUE;
    * (long *) (bloc + taille + 3 * sizeof(long)) = VALEUR_MAGIQUE;
    /* on renvoie un pointeur sur le bloc réservé à l'appelant */
    return (bloc + 2 * sizeof(long));
}

    static void
mon_free (void * pointeur, void * appel)
{
    long taille;
    long i;

    if (! verifie_pointeur(pointeur)) {
        fprintf(stderr, "%p : free avec mauvais bloc\n", appel);
        abort();
    }
    if (pointeur == NULL)
        return;
    RESTITUTION_POINTEURS();
    /* écrabouillons les données ! */
    taille = (* (long *) (pointeur - 2 * sizeof(long)));
    for (i = 0; i < taille + 4 * sizeof(long); i++)
        * (char *) (pointeur - 2 * sizeof(long) + i) = 0;
```

```
    /* et libérons le pointeur */
    free(pointeur - 2 * sizeof(long));
    SAUVEGARDE_POINTEURS();
    INSTALLATION_ROUTINES ();
}

    static int
verifie_pointeur (void * pointeur)
{
    long    taille;

    if (pointeur == NULL)
        return 1;
    if (* (long *) (pointeur - sizeof(long)) != VALEUR_MAGIQUE)
        return 0;
    taille = * (long *) (pointeur - 2 * sizeof(long));
    if (* (long *) (pointeur + taille) != VALEUR_MAGIQUE)
        return 0;
    if (* (long *) (pointeur + taille + sizeof(long)) != VALEUR_MAGIQUE)
        return 0;
    return 1;
}
```

Nous avons défini des macros pour remplacer les trois lignes de copie de pointeurs pour raccourcir et simplifier le fichier source. Voici un exemple d'exécution avec détection du débordement de la chaîne :

```
$ ./exemple_hook_2
0x804867b : free avec mauvais bloc
Aborted (core dumped)
$ rm core
$
```

Bien entendu, ce genre de routine ne permet pas de détecter tous les défauts de gestion mémoire dans un programme, mais on peut l'associer avec les autres méthodes, comme `mtrace()`, pour vérifier que l'application est aussi robuste qu'elle en a l'air extérieurement.

Conclusion

Ce chapitre a permis de mettre en place les principes fondamentaux de la gestion de la mémoire d'un processus. Il s'agit de notions élémentaires et de routines présentes dans l'essentiel des applications courantes. Les méthodes de débogage ou de paramétrage présentées ici sont particulièrement précieuses, mais elles ne sont malheureusement pas portables en dehors d'un environnement de programmation Gnu. Le prochain chapitre va nous permettre d'aborder des notions plus pointues sur la manipulation de l'espace mémoire d'un processus.

Les fonctions générales d'allocation mémoire sont décrites dans [KERNIGHAN 1994] *Le langage C*. Pour obtenir des précisions sur les rapports entre les fonctions de bibliothèque comme `malloc()` et les appels-système comme `brk()` ou `mmap()`, on consultera avec profit les sources de la bibliothèque GlibC. On notera également qu'une discussion concernant les risques induits par les allocations dynamiques et que des idées pour dépister les bogues sont présentées dans [MCCONNELL 1994] *Programmation professionnelle*.

14

Gestion avancée de la mémoire

Nous nous intéresserons dans ce chapitre à divers aspects de la gestion mémoire, plus complexes que les allocations et libérations étudiées dans le précédent chapitre. La plupart de ces fonctions ont été introduites par la norme Posix.1b (avant d'être intégrées dans SUSv3) car elles concernent souvent les applications temps-réel.

Nous étudierons tout d'abord le mécanisme du verrouillage de pages en mémoire, qui intéresse principalement les processus temps-réel et les applications de cryptographie.

Nous verrons par la suite les possibilités offertes par le noyau Linux en ce qui concerne la projection de fichier dans une zone de l'espace mémoire du processus. Nous pourrons alors comprendre le détail des techniques d'allocation de mémoire dynamique au niveau du noyau.

Enfin, nous observerons les protections d'une page mémoire contre les accès indésirables.

Verrouillage de pages en mémoire

La gestion de la mémoire virtuelle sous Linux permet aux processus de disposer, dans leur ensemble, de beaucoup plus de mémoire que la machine n'en contient physiquement. Nous en avons vu une illustration dans les premiers paragraphes du chapitre précédent, où le programme `exemple_malloc_2` arrivait à disposer de 202 Mo sur une machine ne comportant que 128 Mo de mémoire vive.

Lorsque le noyau doit allouer une nouvelle page et qu'il n'en a plus de disponible en mémoire vive, il recherche une page inutilisée depuis longtemps et la rejette de la mémoire. Si cette page contient uniquement des données qui ont été lues depuis un fichier, par exemple le code exécutable d'un programme, elle est purement et simplement effacée. On sait où la retrouver. Si la page a été modifiée par le processus auquel elle appartient, par contre le noyau la sauvegarde temporairement sur son périphérique de swap.

Un processus est donc en permanence susceptible de voir l'une de ses pages disparaître temporairement dans la mémoire secondaire, pour être rechargée lorsqu'il en demandera l'accès.

Ce mécanisme très utile est transparent pour les processus, mais peut présenter néanmoins plusieurs inconvénients dans certains cas bien précis :

- Au moment où un processus tente d'accéder à une zone mémoire qui a été rejetée sur le périphérique de swap, le processeur déclenche une faute de page. Le noyau recharge alors la page manquante et rend le contrôle au périphérique. Pour trouver suffisamment de place pour charger la page en question, le noyau peut être obligé d'évacuer d'autres zones de données de la mémoire, quitte à les sauvegarder à leur tour sur le disque. Tout cela ralentit considérablement l'accès à la zone mémoire, et pire, le ralentissement n'est pas prévisible. Or, sur des systèmes de contrôle industriel par exemple, des processus séquencés suivant un mécanisme temps-réel doivent pouvoir accéder à leur mémoire dans un temps limité, pas nécessairement de façon extrêmement rapide mais constante, quelles que soient les conditions de fonctionnement de la machine.
- Il peut arriver que des processus contiennent temporairement des données hautement confidentielles, comme des mots de passe ou des clés de cryptage. Si le noyau bascule sur le périphérique de swap la zone contenant ces données, elles s'y trouveront encore après le rechargement de la page en mémoire vive. Il est alors possible de démonter le périphérique de swap assez rapidement et de l'examiner octet par octet pour rechercher les informations secrètes.

Le noyau Linux offre une solution à ces problèmes, ou plutôt deux solutions différentes. Dans le premier cas, notre processus contient des zones critiques où il ne peut pas se permettre le moindre délai imprévu. Aucune zone de mémoire ne doit donc être projetée sur le swap. De plus, le processus doit disposer de suffisamment d'espace de pile libre pour les invocations des sous-routines. Rappelons que nous considérons uniquement les zones critiques de notre processus temps-réel, zones dans lesquelles on ne s'amusera pas à faire des allocations de nouvelles pages de mémoire, par exemple. Nous devons donc avoir l'assurance à l'entrée dans la portion critique qu'aucune partie du processus (données, code, pile) ne sera évacuée sur le périphérique de swap, et ceci jusqu'à la sortie de la section critique.

Pour le second cas par contre, seules de petites parties de la mémoire doivent rester absolument en mémoire physique. Le code, la pile et les autres données du processus peuvent très bien être éjectés temporairement ; nous voulons seulement nous assurer que nos données confidentielles ne seront pas stockées, même passagèrement sur le disque. Bien entendu, ce genre de processus doit utiliser l'appel-système `setrlimit()` pour mettre à zéro sa limite `RLIMIT_CORE`, comme nous l'avons vu dans le chapitre sur l'exécution des processus, afin d'être sûr de ne pas laisser une image mémoire sur le disque en cas de terminaison anormale.

L'appel-système **`mlock()`** permet de verrouiller une zone particulière dans la mémoire centrale. Les données qui s'y trouvent ne seront pas éjectées sur le swap avant que le processus ne se termine, qu'il effectue un `exec()` ou qu'il invoque l'un des appels-système **`munlock()`** et **`munlockall()`**.

L'appel-système **`mlockall()`** verrouille toute la mémoire appartenant au processus, aussi bien son espace de code, de données, que sa pile, les bibliothèques partagées qu'il utilise ou les fichiers projetés que nous verrons à la prochaine section.

L'appel `munlock()` déverrouille uniquement une zone précise de mémoire alors que `munlockall()` déverrouille toute la mémoire du processus.

Le verrouillage de page en mémoire est une opération privilégiée car elle lèse nécessairement les autres processus qui disposeront de moins de mémoire physique à se partager. Ces appels-

système sont donc réservés aux processus s'exécutant avec l'UID effectif de *root* ou possédant la capacité `CAP_IPC_LOCK`. Notez que le nom de cette capacité évoluera peut-être dans l'avenir car le préfixe *IPC* représente habituellement les communications entre processus et est surtout utilisé avec le mécanisme IPC Système V que nous verrons dans le chapitre 25.

Il faut bien comprendre qu'il n'y a pas d'empilement des verrouillages. Autrement dit, une zone qui a été verrouillée plusieurs fois, par exemple en intersection de plusieurs `mlock()` et avec un `mlockall()`, sera déverrouillée avec un seul appel `munlock()`. Il ne faut donc pas utiliser des verrouillages temporaires dans des routines sans être conscient qu'on risque de déverrouiller des pages que la fonction appelante considérait comme fixées.

Les prototypes des fonctions de verrouillage sont déclarés dans `<sys/mman.h>` ainsi :

```
int mlock (const void * adresse_debut, size_t longueur)
int munlock (const void * adresse_debut, size_t longueur);
int mlockall (int type);
int munlockall (void);
```

Tous ces appels-système renvoient 0 s'ils réussissent et –1 en cas d'échec.

On peut préciser en argument de `mlockall()` ce qu'on désire verrouiller, en effectuant un OU binaire entre les constantes suivantes :

`MCL_CURRENT` permet de verrouiller les zones mémoire actuellement possédées par le processus.

`MCL_FUTURE` sert à verrouiller les zones qui appartiendront au processus dans l'avenir. Ceci concerne bien entendu les nouvelles zones de mémoire allouées dynamiquement, mais également la pile par exemple.

> **Attention**
> Lorsqu'on veut verrouiller l'ensemble des pages actuelles et celles qui seront allouées dans l'avenir, il faut utiliser `MCL_CURRENT |MCL_FUTURE` et non `MCL_FUTURE` seule.

Avant d'entrer dans une section critique d'un processus temps-réel, nous devons allouer toutes les données nécessaires, les verrouiller en mémoire, et réserver assez de place dans la pile pour tous les appels de sous-routines durant cette portion critique. On utilisera un schéma comme celui-ci :

```
/*
 * Allocation de toutes les variables dynamiques nécessaires
 */
    ... malloc(...) ...

/*
 * Écriture dans les variables dynamiques pour être sûr
 * qu'elles sont effectivement en mémoire
 */
 ... memset(...) ...

/*
 * Réservation de l'espace de pile, et verrouillage par
 * une fonction spéciale.
 */
```

```
  if (reserve_pile_et_verrouille() != 0) {
      perror("mlockall");
      exit(EXIT_FAILURE);
  }

  /*
   * Maintenant nous pouvons entrer dans la section critique
   */
       ...
       ...
       ...
  /*
   * Sortie de la portion critique du code
   */

  munlockall();

/*
 * Voici à présent la routine réservant la place nécessaire dans
 * la pile et verrouillant la mémoire
 */

#define TAILLE_RESERVEE   8192

    static int
reserve_pile_et_verrouille (void)
{
    char * reserve;

    /* réservation dans la pile */
    reserve = alloca(TAILLE_RESERVEE);

    /* écriture pour s'assurer que les pages sont bien affectées */
    memset(reserve, 0, TAILLE_RESERVEE);

    /* verrouillage de la mémoire */
    return mlockall(MCL_CURRENT);

    /* en revenant, la mémoire occupée par la variable reserve est libérée
     * mais les pages restent en mémoire physique à la disposition du
     * processus pour sa pile.
     */
}
```

Sous Linux, on ne peut verrouiller au maximum que la moitié de la mémoire physique du système. Les appels-système `mlock()` et `mlockall()` peuvent donc échouer si on essaye de dépasser cette quantité (où s'il n'y a plus assez de place en mémoire à cause d'un autre processus ayant verrouillé d'autres zones). Cela signifie aussi que lorsqu'on utilise `MCL_FUTURE` avec `mlockall()`, des allocations mémoire à venir pourront échouer à cause de cette limitation.

Le noyau verrouillant des pages entières de mémoire, on risque des chevauchements entre des zones mémoire proches. Il est en effet possible de déverrouiller entièrement une page alors

qu'elle contient des variables qu'on croit encore protégées. Dans le cas d'un processus désirant protéger des données bien précises mais pas tout son espace mémoire, il est conseillé de laisser toutes les données verrouillées jusqu'à ce qu'elles puissent toutes être relâchées sans risque.

Le verrouillage d'une zone de mémoire n'est pas hérité au cours d'un `fork()`. Les données restent verrouillées en mémoire pour le père, mais dès que l'un des deux processus veut modifier le contenu d'une page mémoire, le noyau en fait une copie à l'intention du fils, et cette copie n'est pas verrouillée. On peut donc dire qu'en cas de verrouillage complet avec `mlockall()`, le processus fils bénéficiera probablement du maintien en mémoire du code du programme jusqu'à la fin de son père, mais ce n'est pas obligatoire.

Les fonctions de verrouillage de pages en mémoire sont définies par SUSv3. Pour assurer la portabilité d'un programme, on prendra en compte dans la compilation, par des tests conditionnels, les constantes symboliques :

- **`_POSIX_MEMLOCK`** qui indique que `mlockall()` et `munlockall()` sont utilisables sur le système.
- **`_POSIX_MEMLOCK_RANGE`** qui indique que `mlock()` et `munlock()` sont disponibles.

Projection d'un fichier sur une zone mémoire

Il est parfois intéressant, plutôt que de travailler directement sur le contenu d'un fichier, d'en projeter une image en mémoire, sur laquelle on œuvre ensuite comme avec des variables normales. Cette projection permet de manipuler le contenu du fichier au moyen d'une image en mémoire, donc beaucoup plus rapidement et simplement qu'avec de véritables lectures et écritures sur le disque.

Le noyau offre l'appel-système **`mmap()`** pour assurer la projection d'un fichier en mémoire. Son prototype est déclaré dans `<sys/mman.h>` ainsi :

```
void * mmap (void * debut, size_t longueur,
             int protection, int attribut,
             int fd, off_t decalage);
```

Nous allons voir les arguments les uns après les autres. Précisons tout de suite qu'en cas d'échec, `mmap()` renvoie la constante symbolique `MAP_FAILED`, sinon il renvoie un pointeur sur la zone de mémoire allouée pour la projection. Cette zone se trouve dans la portion de l'espace d'adressage nommée « tas ». Elle se situe au-dessus des données globales non initialisées, et en dessous de la pile.

L'argument `début` indique l'emplacement dans lequel on désire effectuer la projection. Il ne s'agit que d'un simple désir, `mmap()` pouvant utiliser n'importe quelle autre adresse s'il le préfère. En général, on ne s'occupe pas de cet argument, on transmet `NULL` pour signifier à `mmap()` de prendre l'emplacement de son choix. Notons qu'on ne doit pas allouer de mémoire pour la projection ; en fait la zone de projection ne doit même pas avoir d'intersection avec un bloc mémoire précédemment alloué. Si on désire absolument choisir un emplacement précis, il faut passer une adresse alignée sur une frontière de page. La taille d'une page est disponible en interrogeant la fonction `sysconf()` avec l'argument `_SC_PAGESIZE`. Nous ne développerons pas plus l'utilisation de l'argument `début` car il est principalement utilisé par les développeurs du noyau pour le chargement des fichiers exécutables et des librairies partagées, sans intérêt immédiat pour un programmeur applicatif.

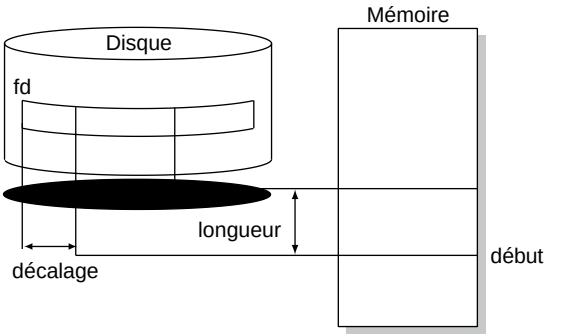


Figure 14.1
Projection d'un fichier en mémoire

L'argument `longueur` indique, on s'en doute, la longueur de la projection en mémoire. Cet argument sera arrondi par excès pour contenir un nombre entier de pages mémoire. On peut indiquer plusieurs types de `protection` dans le troisième argument, en utilisant un OU binaire entre les constantes suivantes :

- **`PROT_EXEC`** : le processus pourra exécuter le code contenu dans la projection ainsi réalisée. Ceci est utilisé par le noyau pour implémenter l'appel-système `exec()` avec les fichiers binaires exécutables.
- **`PROT_READ`** : les pages obtenues seront accessibles en lecture.
- **`PROT_WRITE`** : on pourra écrire dans la projection. Le noyau assurera la synchronisation avec le fichier disque dans des circonstances dont nous reparlerons plus bas.

En fait, il existe également la protection vide **`PROT_NONE`** grâce à laquelle on ne peut ni lire, ni écrire, ni exécuter de code dans la mémoire projetée. On peut l'utiliser de manière très spécifique pour le débogage, comme nous le décrirons dans la section traitant des protections de zones mémoire, mais c'est très rare.

> **Attention**
> Il faut remarquer que sur les architectures x86, le fait de demander `PROT_WRITE` entraîne obligatoirement `PROT_READ`, et qu'il n'y a pas de différence entre `PROT_READ` et `PROT_EXEC`. Ce n'est toutefois pas le cas sur d'autres processeurs, aussi positionnera-t-on bien les protections dont on a besoin.

L'argument suivant, `attribut`, est composé d'un OU binaire entre les constantes symboliques suivantes :

- **`MAP_PRIVATE`** : la projection n'est pas destinée à être réécrite sur le disque, et la zone de mémoire allouée ne doit pas être partagée avec d'autres processus. Si le programme effectue une modification sur la zone de projection, une copie privée de cette portion de mémoire lui sera attribuée.
- **`MAP_SHARED`** : au contraire de l'attribut précédent, les écritures dans la zone de projection sont destinées à être vues par le reste du monde. Dans le cas où la zone de mémoire est partagée, comme nous le verrons dans les communications entre processus, toute modification est immédiatement perceptible dans les autres programmes. Par contre, la synchronisation effective avec le fichier disque n'est pas nécessairement immédiate. Il faut être

prudent si un processus concurrent tente d'accéder directement au fichier car il ne verra pas forcément les mêmes données que celles de la projection en mémoire.

- **MAP_FIXED** : permet de signaler à `mmap()` qu'on désire absolument disposer d'une projection à l'adresse de départ indiquée en premier argument. Si ce n'est pas possible, l'appel-système échouera.

Ces attributs sont les seuls qui sont définis par SUSv3, mais Linux propose également d'autres attributs spécifiques, en voici quelques-uns :

- **MAP_ANON** ou **MAP_ANONYMOUS** : on ne projette pas réellement un fichier, mais on désire obtenir une zone mémoire vierge, emplie de zéros. La bibliothèque GlibC l'utilise pour implémenter `malloc()`. Lorsque cet attribut est mentionné, les deux derniers arguments `fd` et `decalage` de l'appel-système sont ignorés. Sur les systèmes où `MAP_ANONYMOUS` n'existe pas, on peut se servir des pseudo-fichiers `/dev/null` ou `/dev/zero` pour allouer une zone de mémoire vierge.
- **MAP_DENYWRITE** : empêche l'écriture dans le fichier correspondant tant que la projection en mémoire est valide. C'est ce qu'utilise l'appel-système `exec()` pour verrouiller le fichier exécutable correspondant au processus lancé. Cela permet au noyau de supprimer des pages de code pour libérer de la mémoire tout en étant sûr de pouvoir les relire par la suite. C'est aussi ce qui déclenche une erreur `ETXTBSY` de l'éditeur de liens lorsqu'on relance une compilation sans avoir arrêté le processus résultant. Malheureusement, cet attribut est silencieusement ignoré durant les appels-système, seul le noyau a la possibilité de l'utiliser.
- **MAP_GROWSDOWN** : indique qu'on utilise une projection anonyme pour allouer une zone de mémoire qui grossira par le bas. Cet attribut sert au noyau pour mettre en place la pile d'un processus.

Lors d'un appel à `mmap()`, il *faut* utiliser l'une des deux constantes `MAP_PRIVATE` ou `MAP_SHARED`, même si elles n'ont de sens que lorsqu'on écrit dans la zone de projection. Toutes les autres sont facultatives.

On comprend mieux à présent comment `mmap()` est utilisé par `malloc()` pour réserver de gros blocs en mémoire, et pourquoi la mémoire allouée avec `calloc()` contient des zéros sans qu'il y ait eu de véritable écriture pour l'initialiser : lorsque nous demandons une projection `PRIVATE` et `ANONYMOUS`, le noyau nous fournit des pages virtuelles, sans correspondance réelle avec la mémoire physique du système. Quand nous essayons de lire le contenu de ces zones, une faute de page est déclenchée et le noyau, s'apercevant qu'il s'agit d'une projection anonyme (ou du pseudo-fichier `/dev/zero`), sait qu'il doit nous renvoyer des zéros. Par contre, dès que nous tentons d'écrire sur cette projection, la faute de page qui se déclenche oblige le noyau à affecter une véritable zone de mémoire physique au processus, à copier le contenu de la projection (en l'occurrence il suffit de remplir la nouvelle page avec des zéros), puis à autoriser l'écriture par le programme. Ce n'est donc qu'au moment de l'écriture dans la mémoire allouée avec `mmap()` que le stock de pages physiquement disponibles diminue.

L'argument `fd` correspond à un descripteur de fichier déjà ouvert par le processus. Bien entendu le mode d'ouverture devra correspondre, en ce qui concerne les possibilités d'écriture dans le fichier, aux protections de la zone mémoire. Dans tous les cas, le fichier devra permettre la lecture, autrement dit il faudra l'ouvrir en mode `O_RDONLY` ou `O_RDWR` (nous verrons la signification de ces modes dans le chapitre consacré à la gestion des fichiers).

Le dernier argument, `decalage`, correspond à la position dans le fichier de la projection. On n'est pas obligé en effet de projeter tout le fichier, pas plus d'ailleurs que le début.

Le décalage est mesuré en nombre d'octets, comme la valeur qu'on transmet à un appel-système `lseek()`.

Lorsque `mmap()` échoue, la variable globale `errno` est mise à jour et contient un code indiquant l'erreur qui s'est produite. Certaines erreurs ne sont pas vraiment intuitives, aussi nous allons les détailler rapidement :

Erreur	Problèmes
`EACCES`	Le fichier ne nous permet pas la lecture. Le fichier ne nous permet pas l'écriture, et on a réclamé un accès à la fois en `PROT_WRITE` sur la zone et un partage `MAP_SHARED`. Dans le cas d'une projection privée `MAP_PRIVATE`, ce n'est pas un problème, le fichier ne sera pas mis à jour en cas de modification de la zone de mémoire puisqu'on travaille sur une copie. En projection partagée, il faut pouvoir inscrire sur le disque les modifications pour les rendre visibles à l'extérieur. De même, en projection partagée, un fichier ne doit pas être ouvert en mode d'ajout uniquement en fin de fichier.
`EAGAIN`	Le processus a demandé un verrouillage en mémoire de ses allocations à venir avec l'argument `MCL_FUTURE` de l'appel `mlockall()`. Allouer la taille demandée dépasserait sa limite de mémoire verrouillée `RLIMIT_MEMLOCK`, accessible avec `getrlimit()`. Un verrouillage strict a été placé par le processus avec `fcntl()` sur le fichier, et on demande une projection partagée avec `MAP_SHARED`.
`EINVAL`	On a demandé une projection à une adresse invalide ou de longueur beaucoup trop grande. L'adresse mentionnée n'est pas alignée sur une frontière de page alors qu'on a demandé une projection figée avec `MAP_FIXED`. On n'a utilisé ni l'attribut `MAP_SHARED` ni `MAP_PRIVATE`, ou au contraire on les a mentionnés tous les deux. On a demandé une projection `ANONYMOUS` avec un attribut `MAP_SHARED`. C'est impossible car la projection anonyme nécessite de faire une copie dès une tentative d'écriture dans les pages virtuelles vierges. Attention, sur certains systèmes ce type de projection est autorisée et permet de créer de la mémoire partagée entre un processus père et son fils. Sous Linux, ce n'est pas le cas. Toutefois on peut utiliser un fichier temporaire de la dimension voulue pour partager un bloc mémoire facilement, comme nous le ferons dans notre second exemple.
`ENODEV`	Le fichier mentionné ne permet pas la projection en mémoire. Ceci concerne surtout des fichiers spéciaux représentant des périphériques comme `/dev/tty` par exemple.
`ENOMEM`	Il y a trop de projections en mémoire. La limite `MAX_MAP_COUNT` du nombre de projections simultanées sur le système est définie dans `<linux/sched.h>`. Elle vaut 65536 sur les architectures x86. On ne peut pas obtenir une zone de mémoire de la taille demandée. Ceci peut aussi correspondre à un manque de mémoire du noyau le rendant incapable de mettre à jour ses structures internes.

L'erreur `ETXTBSY` devrait normalement être renvoyée si on demande la projection en mémoire avec l'attribut `MAP_DENYWRITE` d'un fichier déjà ouvert en écriture par plusieurs processus. Toutefois, cet attribut est silencieusement effacé par le noyau Linux lors de l'appel-système.

Une fois que `mmap()` a réussi, il est possible de refermer le fichier, la projection en mémoire persistera jusqu'à ce qu'on la libère avec `munmap()`. Les zones projetées sont automatiquement libérées lors de la fin d'un processus ou d'un appel `exec()`. Par contre, les projections sont héritées au cours d'un `fork()`, nous en verrons un exemple plus loin. Les projections `PRIVATE` sont copiées au moment de la première tentative d'écriture dans la zone, aussi il n'y a aucune ambiguïté entre les processus père et fils.

L'appel-système **munmap()** a le prototype suivant :

```
int munmap (void * zone, size_t longueur);
```

Il libère la projection se trouvant dans la zone indiquée, avec la longueur passée en second argument. La seule manière raisonnable d'utiliser munmap() est de lui transmettre le pointeur fourni par un appel antérieur à mmap(), ainsi que la même longueur que celle qu'on avait passée à mmap(). Lorsqu'il réussit, munmap() renvoie 0. La zone de mémoire est alors libérée et rendue inutilisable.

On invoque rarement munmap(), car la plupart du temps les projections qu'on établit en mémoire sont utilisées pendant toute la durée de vie du processus, et on laisse le système les libérer automatiquement à la terminaison du programme.

Il est possible de faire une projection d'un fichier complet, dont la taille dépasse largement celle de la mémoire totale – physique et swap – du système. En effet, le noyau libérera des pages mémoire au fur et à mesure, en les réécrivant sur le disque si elles ont été modifiées, ou en les effaçant purement et simplement si elles sont intactes. La limitation théorique est celle de l'espace d'adressage disponible, c'est-à-dire sur architecture x86 4 Go moins 1 Go réservé au noyau, moins le code du processus, sa pile et ses données, ce qui nous laisse quand même normalement plus de 2 Go. Attention, ceci n'est vrai que si le noyau peut éliminer de la mémoire les pages qui l'encombrent, autrement dit le calcul est totalement différent si on utilise une projection PRIVATE et si on modifie les données projetées. Dans ce cas, la limite de projection est de l'ordre de la taille de la mémoire virtuelle disponible.

Notre premier exemple va consister à écrire un programme qui retourne complètement un fichier, en échangeant successivement le premier octet et le dernier, le deuxième et l'avant-dernier, et ainsi de suite. L'intérêt d'un tel utilitaire reste encore à démontrer, mais on pourrait très bien le transformer pour effectuer du traitement d'image, par exemple.

Nous utiliserons la fonction stat() – que nous étudierons ultérieurement en détail – pour connaître la taille du fichier, puis nous le projetterons en mémoire et nous le retournerons simplement en le considérant comme un tableau de caractères. Ceci aurait été particulièrement fastidieux à écrire avec des appels-système read() et write(), alors que l'implémentation se fait, avec mmap(), de manière très intuitive.

exemple_mmap_1.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/mman.h>

    int
main (int argc, char * argv[])
{
    char *      projection;
    int         fichier;
    struct stat etat_fichier;
    long        taille_fichier;
    int         i;
    char        tmp;
```

```
    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier_a_inverser\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fichier = open(argv[1], O_RDWR)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    if (stat(argv[1], & etat_fichier) != 0) {
        perror("stat");
        exit(EXIT_FAILURE);
    }
    taille_fichier = etat_fichier.st_size;
    projection = (char *) mmap(NULL, taille_fichier,
                PROT_READ | PROT_WRITE, MAP_SHARED, fichier, 0);
    if (projection == (char *) MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fichier);
    for (i = 0; i < taille_fichier / 2; i ++) {
        tmp = projection[i];
        projection[i ] = projection[taille_fichier - i - 1];
        projection[taille_fichier - i - 1] = tmp;
    }
    munmap((void *) projection, taille_fichier);
    return EXIT_SUCCESS;
}
```

Nous l'utilisons pour retourner un petit fichier de texte :

```
$ cat > test.txt

AZERTYUIOP
QSDFGHJKLM
WXCVBN
  (Contrôle-D)
$ ./exemple_mmap_1 test.txt
$ cat test.txt


NBVCXW
MLKJHGFDSQ
POIUYTREZA
$ ./exemple_mmap_1 test.txt
$ cat test.txt

AZERTYUIOP
QSDFGHJKLM
WXCVBN

$
```

À présent, nous allons vérifier que, comme nous l'annoncions plus tôt, il est possible de projeter en mémoire un fichier plus gros que l'ensemble de la mémoire virtuelle si on utilise bien une projection `SHARED`, ce qui est le cas dans notre exemple. Pour cela, nous avons besoin d'un gros fichier de test. Nous allons donc écrire un petit programme créant un fichier contenant le nombre de méga-octets qu'on demande en ligne de commande. Chaque bloc d'un méga-octet sera rempli avec une valeur différente pour vérifier que le retournement se passe bien.

cree_gros_fichier.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    #define TAILLE_BLOC  (1024 * 1024)

    int
main (int argc, char * argv[])
{
    int    nombre_blocs;
    FILE * fp;
    char * bloc;
    int    i;

    if ((argc != 3) || (sscanf(argv[2], "%d", & nombre_blocs) != 1)) {
        fprintf(stderr, "Syntaxe : %s fichier nb_blocs \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fp = fopen(argv[1], "w")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    if ((bloc = malloc(TAILLE_BLOC)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    for (i = 0 ; i < nombre_blocs; i ++) {
        memset(bloc, i, TAILLE_BLOC);
        if (fwrite(bloc, 1, TAILLE_BLOC, fp) != TAILLE_BLOC) {
            perror("fwrite");
            exit(EXIT_FAILURE);
        }
    }
    fclose(fp);
    return EXIT_SUCCESS;
}
```

Pour observer le contenu du fichier, on utilisera le petit utilitaire `exemple_getchar.c` que nous avions développé dans le chapitre 10. Bien entendu, nous ne présenterons ici que des extraits de son affichage.

La mémoire totale de notre système fait 256 Mo, c'est donc la valeur que nous choisirons pour notre gros fichier (et qui nous évite un retour à zéro des caractères de remplissage après avoir dépassé 255).

Ces programmes sont assez gourmands en ressources système, il vaut donc mieux éviter de les faire fonctionner sur une machine ayant plusieurs utilisateurs, ou alors il faut les lancer avec la commande `nice`.

```
$ ./cree_gros_fichier test.bin 256
$ ls -l test.bin
-rw-r--r--   1 ccb      ccb      268435456 Oct 10 18:28 test.bin
$ ../10/exemple_getchar < test.bin
00000000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
  [...]
000FFFE0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
000FFFF0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
00100000  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
00100010  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
  [...]
001FFFE0  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
001FFFF0  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
00200000  02 02 02 02 02 02 02 02-02 02 02 02 02 02 02 02
00200010  02 02 02 02 02 02 02 02-02 02 02 02 02 02 02 02
  [...]
  [...]
0FFFFFC0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
0FFFFFD0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
0FFFFFE0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
0FFFFFF0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
$ ./exemple_mmap_1 test.bin
$ ../10/exemple_getchar < test.bin
00000000  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
00000010  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
  [...]
000FFFE0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
000FFFF0  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF
00100000  FE FE FE FE FE FE FE FE-FE FE FE FE FE FE FE FE
00100010  FE FE FE FE FE FE FE FE-FE FE FE FE FE FE FE FE
  [...]
  [...]
0FEFFFE0  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
0FEFFFF0  01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01
0FF00000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
0FF00010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
  [...]
0FFFFFE0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
0FFFFFF0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
$ rm test.*
$
```

Bien sûr, la vérification du fonctionnement de notre programme est un peu sommaire car le contenu de chaque bloc d'un méga-octet est constant, mais nous avons bien manipulé, directement dans notre espace d'adressage, un fichier dont la taille dépasse largement la mémoire que le système nous offre. L'intérêt de ce genre de projection apparaît plus clairement dans les domaines du traitement d'images ou des échantillons de son numérique, qu'on manipule

ainsi comme des blocs de mémoire même si le fichier sous-jacent dépasse, de loin, la mémoire virtuelle du système.

Notre second exemple d'utilisation de `mmap()` va consister à utiliser la projection partagée d'un fichier temporaire pour disposer d'une variable partagée entre un processus et son fils. Nous désirons obtenir une zone mémoire de la taille (au moins) d'un entier. Nous allons demander au système de nous donner le nom d'un fichier temporaire, puis nous allons le créer et y écrire une variable de la taille désirée. Ensuite, nous projetterons ce fichier en mémoire, et nous utiliserons l'adresse résultante comme pointeur sur une variable entière.

Les deux processus peuvent alors se séparer. Rappelons que les projections en mémoire sont héritées lors d'un `fork()`. Le processus père va incrémenter cette variable de 0 à 9, en envoyant après chaque mise à jour un signal `SIGUSR1` à son fils, et en dormant pendant une seconde (méthode sale mais simple pour attendre que le fils ait affiché son résultat). Le processus fils ne fait qu'écrire à l'écran le contenu de la variable entière au sein de son gestionnaire de signaux.

exemple_mmap_2.c :

```
    #include <fcntl.h>
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/mman.h>

    int * entier;

    void
gestionnaire_sigusr1 (int num)
{
    fprintf(stdout, "Fils : * entier = %d\n", * entier);
    fflush(stdout);
}

    int
main (void)
{
    char * nom_fichier;
    int    fichier;
    pid_t  pid;

    if (signal(SIGUSR1, gestionnaire_sigusr1) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    if ((nom_fichier = tmpnam(NULL)) == NULL) {
        perror("tmpnam");
        exit(EXIT_FAILURE);
    }
    if ((fichier = open(nom_fichier, O_RDWR | O_CREAT | O_TRUNC,
                    S_IRUSR | S_IWUSR)) < 0) {
```

```
        perror("open");
        exit(EXIT_FAILURE);
    }
    if (write(fichier, & fichier, sizeof(int)) != sizeof(int)) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    entier = (int *) mmap(NULL, sizeof(int),
                        PROT_READ | PROT_WRITE, MAP_SHARED,
                        fichier, 0);
    if (entier == (int *) MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fichier);
    unlink(nom_fichier);

    if ((pid = fork()) < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        while (1)
            sleep(1);
    } else {
        for ((* entier) = 0; (* entier) < 10; (* entier) ++) {
            fprintf(stdout, "Père : * entier = %d\n", * entier);
            fflush(stdout);
            kill(pid, SIGUSR1);
            sleep(1);
        }
        /* Ne pas oublier de tuer le fils qui est en attente */
        kill(pid, SIGKILL);
    }
    return EXIT_SUCCESS;
}
```

On notera qu'on détruit avec `unlink()` le fichier temporaire après l'avoir refermé. Le noyau ne l'effacera toutefois complètement que lorsque sa dernière référence sera refermée, en l'occurrence lorsque la fin des processus libérera la zone de projection en mémoire.

```
$ ./exemple_mmap_2
Père : * entier = 0
Fils : * entier = 0
Père : * entier = 1
Fils : * entier = 1
Père : * entier = 2
Fils : * entier = 2
Père : * entier = 3
Fils : * entier = 3
Père : * entier = 4
Fils : * entier = 4
```

```
Père : * entier = 5
Fils : * entier = 5
Père : * entier = 6
Fils : * entier = 6
Père : * entier = 7
Fils : * entier = 7
Père : * entier = 8
Fils : * entier = 8
Père : * entier = 9
Fils : * entier = 9
$
```

Nous voyons ainsi une méthode assez simple et amusante de partager de la mémoire entre deux processus père et fils. Bien entendu, nous observerons une autre technique, plus conventionnelle, dans le chapitre consacré aux communications entre processus.

Dans notre dernier exemple, nous ne nous sommes pas soucié du contenu effectif du fichier projeté. La seule chose qui nous intéressait était que la zone mémoire de la projection soit partagée avec le processus fils. Parfois au contraire, les modifications apportées au fichier doivent être visibles de l'extérieur, que ce soit pour la lecture directe du fichier ou pour d'autres programmes qui en effectuent une projection dans leur propre mémoire.

Ce n'est, par défaut, que lorsqu'une projection `SHARED` est supprimée avec `munmap()` ou à la fin du processus, que le noyau nous garantit que les modifications apportées à la zone de projection seront répercutées sur le fichier. Si nous désirons nous en assurer à un autre moment, l'appel-système **`msync()`** fournit plusieurs possibilités. Son prototype est le suivant :

```
int msync (const void * debut, size_t longueur, int attribut);
```

Lorsqu'on invoque `msync()`, on lui transmet le pointeur sur la zone de projection qu'on désire mettre à jour, ainsi que la longueur de la zone. Naturellement, il est conseillé de ne demander que la mise à jour des portions effectivement modifiées de la projection, et pas nécessairement tout le fichier. L'attribut fourni en troisième argument peut contenir les constantes symboliques suivantes, liées par un OU binaire :

`MS_ASYNC` : cette option demande au noyau de se préparer à mettre à jour les zones indiquées. Néanmoins, l'écriture n'a pas lieu tout de suite, l'appel-système revenant immédiatement.

`MS_SYNC` : avec cet attribut, le noyau effectue tout de suite la mise à jour. Lorsque l'appel-système revient, nous savons qu'une lecture directe du fichier concerné nous renverrait les informations mises à jour. Toutefois, rien ne nous assure que les données aient été réellement écrites sur le disque, celui-ci peut gérer un cache plus ou moins important et retarder les écritures effectives.

`MS_INVALIDATE` : qu'on utilise une mise à jour synchrone ou asynchrone, le noyau nous assure uniquement qu'une lecture directe du fichier renverra nos données mises à jour. Malgré tout, d'autres processus, totalement indépendants du nôtre, peuvent avoir effectué une projection du même fichier dans leur espace mémoire. Cet attribut garantit que leurs pages seront invalidées et que le noyau les reprendra sur le disque lors du prochain accès aux données.

On ne doit évidemment pas utiliser les options `MS_ASYNC` et `MS_SYNC` simultanément.

Il existe également sous Linux un appel-système supplémentaire. Il n'est disponible qu'en utilisant l'option `_GNU_SOURCE` lors de la compilation. Il s'agit de **`mremap()`**, qui permet à la

manière de `realloc()` d'agrandir ou de rétrécir une zone de projection en mémoire. Son prototype est le suivant :

```
void * mremap (void * zone, size_t ancienne_longueur,
               size_t nouvelle_longueur, int attribut);
```

On transmet en argument un pointeur sur la zone de projection en cours, ainsi que sa longueur, suivi de la nouvelle longueur désirée. L'attribut peut éventuellement prendre comme unique valeur celle de la constante symbolique **`MREMAP_MAYMOVE`**, auquel cas `mremap()` sera autorisé à déplacer la zone de projection dans l'espace d'adressage. L'appel-système renvoie un pointeur sur la nouvelle zone, ou `MAP_FAILED` en cas d'échec. Bien entendu, les relations avec le fichier sous-jacent sont conservées. Si on agrandit une zone de projection, il est possible d'accéder à une plus grande partie du fichier.

Avant de conclure ce chapitre sur la gestion de la mémoire, nous allons nous intéresser au principe de la protection des pages mémoire, sujet que nous avons effleuré sans entrer dans les détails en présentant le troisième argument d'invocation de `mmap()`.

Protection de l'accès à la mémoire

L'appel-système **`mprotect()`** permet de limiter les possibilités d'accès à certaines pages mémoire. Son prototype est le suivant :

```
int mprotect (const void * debut_zone, size_t longueur, int protection);
```

La zone de mémoire à protéger doit être alignée sur une frontière de page. Nous détaillerons tout ceci plus bas.

La protection qu'on réclame en troisième argument est du même genre que celle de l'appel `mmap()`, avec une composition par OU binaire des constantes suivantes :

Constante	Signification
`PROT_NONE`	Aucune autorisation d'accès
`PROT_READ`	Autorisation de lire dans la zone
`PROT_WRITE`	Autorisation d'écrire dans la zone mémoire
`PROT_EXEC`	Possibilité d'y exécuter du code

Lorsque l'appel `mprotect()` réussit, il renvoie 0 et remplace complètement les protections originales de la zone mémoire, sinon il renvoie –1.

L'autorisation `PROT_EXEC` ne concerne normalement pas le programmeur applicatif. De toute manière, sur les architectures x86 par exemple, `PROT_EXEC` et `PROT_READ` ont exactement le même effet. Peut-être y aura-t-il malgré tout une évolution future. Aussi on utilise correctement ces attributs pour assurer la pérennité et la portabilité d'un programme. Il est dommage que `PROT_EXEC` ne soit pas réellement utilisé par la gestion mémoire des processeurs x86, car cela permettrait de déjouer une partie des attaques de sécurité en empêchant formellement d'exécuter des instructions en dehors du segment de code initialisé au chargement du programme. Nous avons déjà vu que de nombreux piratages se basaient sur le débordement de chaînes de caractères locales pour placer des instructions personnelles dans la pile des utilitaires système Set-UID.

Le fait d'interdire tout accès avec l'autorisation vierge PROT_NONE peut servir au débogage d'un programme pour contrôler tous les accès d'une application à une zone de mémoire allouée dynamiquement. Par exemple, on peut placer l'autorisation PROT_NONE dès l'allocation, et ne permettre la lecture qu'à partir du moment où l'initialisation a lieu. Si un autre module tente d'utiliser la variable avant la fin de l'initialisation, le processus sera tué par un signal, et nous pourrons employer gdb et le fichier core pour remonter jusqu'à l'utilisation fautive.

Sur les processeurs x86, le fait de demander une autorisation d'écriture PROT_WRITE entraîne également la disponibilité en lecture (et donc en exécution), mais cela n'est pas nécessairement portable sur les autres architectures.

Nous avons précisé que l'adresse de début de zone doit être alignée sur une frontière de page. C'est automatiquement le cas avec les zones mémoire allouées par mmap() ; aussi est-ce la manière la plus simple d'allouer les zones qu'on protégera ultérieurement. La fonction malloc() utilise en interne l'appel-système mmap() dans certaines conditions, mais on ne peut pas en être certain. S'il y a suffisamment de place libre dans le segment de données qui n'a pas été rendu au système, elle est employée avant toute chose.

Pour assurer la portabilité de notre application, on se conformera donc au standard SUSv3 en allouant les zones mémoire avec mmap() en projection ANONYMOUS. Il faudra simplement se méfier de la différence entre malloc(), qui renvoie NULL en cas d'échec, et mmap(), qui renvoie MAP_FAILED (qui vaut normalement –1). Pour éviter toute ambiguïté, on pourra se redéfinir une fonction d'allocation semblable à malloc(). La seule contrainte est de conserver la taille de la zone allouée car on doit la transmettre à mprotect().

Lorsqu'un processus tente d'accéder de manière illégale à une zone de mémoire protégée, il est tué par le signal SIGSEGV. En voici une illustration :

exemple_mprotect_1.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/mman.h>

    #define TAILLE_CHAINE   128

    void *
mon_malloc_avec_mmap (size_t taille)
{
    void * retour;
    retour = mmap(NULL, taille, PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
    if (retour == MAP_FAILED)
            return NULL;
    return retour;
}

    int
main (void)
{
    char * chaine = NULL;
```

```
    fprintf(stdout, "Allocation de %d octets \n", TAILLE_CHAINE);
    chaine = mon_malloc_avec_mmap(TAILLE_CHAINE);
    if (chaine == NULL) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Protections par défaut \n");
    fprintf(stdout, "   Écriture ...");
    strcpy(chaine, "Ok");
    fprintf(stdout, "Ok\n");
    fprintf(stdout, "   Lecture ...");
    fprintf(stdout, "%s\n", chaine);

    fprintf(stdout, "Interdiction d'écriture \n");
    if (mprotect(chaine, TAILLE_CHAINE, PROT_READ) < 0) {
        perror("mprotect");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "   Lecture ...");
    fprintf(stdout, "%s\n", chaine);
    fprintf(stdout, "   Écriture ...");
    strcpy(chaine, "Non");
    /* ici on doit déjà être arrêté par un signal */
    return EXIT_SUCCESS;
}
```

Nous exécutons le programme, puis nous invoquons `gdb` pour rechercher d'où vient l'erreur :

```
$ ./exemple_mprotect_1
Allocation de 128 octets
Protections par défaut
   Écriture ...Ok
   Lecture ...Ok
Interdiction d'écriture
   Lecture ...Ok
Segmentation fault (core dumped)
$ gdb exemple_mprotect_1 core
GNU gdb 4.17.0.11 with Linux support
  [...]
Core was generated by `./exemple_mprotect_1'.
Program terminated with signal 11, Erreur de segmentation.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0  strcpy (dest=0x40015000 <Address 0x40015000 out of bounds>,
    src=0x8048782 "Non") at ../sysdeps/generic/strcpy.c:38
../sysdeps/generic/strcpy.c:38: Aucun fichier ou répertoire de ce type.
(gdb) bt
#0  strcpy (dest=0x40015000 <Address 0x40015000 out of bounds>,
    src=0x8048782 "Non") at ../sysdeps/generic/strcpy.c:38
#1  0x8048692 in main () at exemple_mprotect_1.c:47
#2  0x40030cb3 in __libc_start_main (main=0x804853c <main>, argc=1,
    argv=0xbffffd14, init=0x8048378 <_init>, fini=0x80486dc <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffffd0c)
```

```
    at ../sysdeps/generic/libc-start.c:78
(gdb) quit
$ rm core
$
```

Le processus est bien tué par un signal `SIGSEGV`, avec création d'une image disque `core`. L'invocation de `gdb` nous apprend que l'erreur s'est produite dans la routine `strcpy()`, dont le fichier source était « `../sysdeps/generic/strcpy.c` » lors de la compilation de la bibliothèque C. Cela ne nous est pas d'un grand secours, aussi utilisons-nous la commande `bt` (pour *backtrace*, suivi en arrière) qui nous permet de voir que `strcpy()` a été invoquée en ligne 47 du fichier `exemple_mprotect_1.c`.

Nous observons que la ligne « Écriture… » n'a pas été transmise sur `stdout`, car le buffer n'a pas été vidé par `fflush()` ni par un retour à la ligne avant l'arrêt brutal du programme.

Le comportement du processeur face à un accès illégal à la mémoire peut paraître déroutant de prime abord. En effet, lorsqu'une faute d'accès est détectée, le noyau est prévenu. Il envoie alors un signal au processus. Par contre, le compteur d'instruction du programme n'est pas incrémenté. Cela signifie que si le processus n'est pas tué par le signal, au retour du gestionnaire de `SIGSEGV`, il retentera la même opération d'accès interdite. En voici une démonstration.

exemple_mprotect_2.c :

```
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/mman.h>

    #define TAILLE_CHAINE   128

void
gestionnaire_sigsegv (int numero)
{
    fprintf(stderr, "Signal SIGSEGV reçu \n");
}

    int
main (void)
{
    char * chaine = NULL;
    if (signal(SIGSEGV, gestionnaire_sigsegv) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Allocation de %d octets \n", TAILLE_CHAINE);
    chaine = mon_malloc_avec_mmap(TAILLE_CHAINE);
    if (chaine == NULL) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Protections par défaut \n");
    fprintf(stdout, "   Écriture ...");
    strcpy(chaine, "Ok");
```

```
    fprintf(stdout, "Ok\n");

    fprintf(stdout, "Interdiction de lecture \n");
    if (mprotect(chaine, TAILLE_CHAINE, PROT_NONE) < 0) {
        perror("mprotect");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "   Lecture ...\n");
    fflush(stdout);
    fprintf(stdout, "%s\n", chaine);
    /* ici on doit déjà être arrêté par un signal */
    return EXIT_SUCCESS;
}
```

Cette fois-ci nous observons l'effet d'une protection `NONE` sur une tentative de lecture. Par contre, nous interceptons le signal `SIGSEGV` et nous affichons simplement un message sur le flux d'erreur standard.

```
$ ./exemple_mprotect_2
Allocation de 128 octets
Protections par défaut
   Écriture ...Ok
Interdiction de lecture
   Lecture ...
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
Signal SIGSEGV reçu
  (Contrôle-C)
$
```

Nous sommes obligé d'arrêter avec Contrôle-C le processus qui part autrement dans un cycle infini d'interception de signal et de tentative de lecture sur une zone verrouillée. Un moyen de se libérer de ce problème serait d'utiliser un saut non local depuis le gestionnaire pour revenir à un emplacement plus sûr du programme. Malgré tout, le fait de détecter une erreur d'accès mémoire doit plutôt être considéré comme un dysfonctionnement à corriger immédiatement à l'aide du fichier d'image `core` et d'un débogueur. Le comportement par défaut de `SIGSEGV` est donc plus approprié.

Conclusion

Nous avons étudié longuement la gestion de la mémoire sous Linux. Ceci nous permet – espérons-le – d'avoir une vision claire des mécanismes mis en œuvre lors des allocations, projections ou accès aux zones de la mémoire tant physique que virtuelle.

Le programmeur applicatif a rarement besoin de se soucier des techniques sous-jacentes aux allocations dynamiques de mémoire. Toutefois, une bonne compréhension de ces phénomènes permet de diagnostiquer plus facilement les problèmes lorsqu'un programme se comporte de manière *a priori* surprenante.

Pour étudier en détail l'espace mémoire d'un processus, les meilleures informations proviennent de l'étude directe des sources du noyau Linux. Toutefois, on trouvera des éléments intéressants dans [BACH 1989] *Conception du système Unix*.

Le prochain chapitre nous permettra d'étudier ce que nous pouvons faire avec les blocs de mémoire ainsi obtenus, y compris les traitements concernant les chaînes de caractères.

15

Utilisation des blocs mémoire et des chaînes

L'emploi des chaînes de caractères et des blocs de mémoire sous Linux n'a rien de particulièrement original par rapport aux autres systèmes d'exploitation. Par contre, la bibliothèque GlibC propose des fonctions très intéressantes, surtout en ce qui concerne les traitements de chaînes. Certaines de ces fonctions sont assez peu connues et recouvrent pourtant des besoins pour lesquels le programmeur est souvent amené à se créer sa propre bibliothèque « maison », alors que l'implémentation de la bibliothèque C est généralement mieux optimisée.

Nous nous intéresserons tout d'abord aux différentes variantes des routines permettant de manipuler des blocs de mémoire bruts sans se préoccuper de leur contenu.

Ensuite, nous verrons les routines de manipulation de chaînes de caractères, notamment celles qui sont utilisées pour mesurer la longueur d'une chaîne, remplir, copier ou comparer des chaînes. Puis, nous nous pencherons sur les fonctions permettant de faire des recherches plus ou moins complexes de sous-chaînes.

Manipulation de blocs de mémoire

Les fonctions essentielles pour manipuler les blocs de mémoire commencent par le préfixe `mem` et sont déclarées dans le fichier `<string.h>`. Sauf mention explicite, les routines présentées ici sont définies dans le standard C. Les extensions Gnu sont accessibles en définissant la constante de compilation `_GNU_SOURCE` avant l'inclusion des fichiers d'en-tête.

Il existe toutefois des routines devenues quasi obsolètes de nos jours mais qu'on peut rencontrer dans d'anciens fichiers source. Nous les mentionnerons pour information, mais il faudra éviter de les utiliser à l'avenir. Avec la GlibC, ces routines sont déclarées dans le fichier `<strings.h>` (avec un `s`).

Les blocs de mémoire qu'on peut manipuler avec la bibliothèque C sont représentés par des pointeurs `void *`. Naturellement, nous déclarerons la plupart du temps nos zones de mémoire avec un autre type, et la conversion sera automatiquement assurée lors de l'invocation des routines. Rappelons que le type générique `void *` peut recevoir n'importe quel type de pointeur sur des données, sans déclencher d'avertissement du compilateur.

Lorsqu'on désire accéder au contenu d'un bloc de mémoire octet par octet, le plus simple est de le déclarer de type `unsigned char *`. Ainsi, nous pourrons adresser directement chaque octet et comparer aisément son contenu avec un entier compris entre 0 et 255. Voici comment manipuler un tel tableau[1] :

```
unsigned char * bloc;
int             i;

if ((bloc = malloc(TAILLE_BLOC)) == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}
for (i = 0; i < TAILLE_BLOC; i ++) {
    bloc[i] = i & 0xFF;
}
```

La première opération qu'on effectue sur un bloc de mémoire est bien souvent de l'initialiser en le remplissant avec une valeur, par ailleurs souvent nulle. La fonction **`memset()`** assure un tel remplissage :

```
void * memset (void * bloc, int valeur, size_t longueur);
```

Le premier argument pointe sur le bloc à remplir. Le second est une valeur entière qui sera convertie en `unsigned char` et qui servira à remplir le nombre d'octets indiqué en troisième argument. `memset()` renvoie la valeur du pointeur `bloc`. L'accès à un bloc de mémoire n'appartenant pas au processus ou le passage d'un pointeur `NULL` peuvent déclencher le signal `SIGSEGV`.

L'emploi de l'ancienne fonction **`bzero()`** est déconseillé. Son prototype était :

```
void bzero (void * bloc, size_t longueur);
```

Elle ne faisait que remplir le bloc avec des zéros. On peut la remplacer par :

```
memset (bloc, 0, longueur).
```

Il est préférable d'utiliser le plus fréquemment possible la fonction `memset()` plutôt que d'essayer d'initialiser manuellement une zone, car cette routine est optimisée, en assurant par exemple sur des processeurs x86 des remplissages directement avec des `long int` pour diviser par 4 le nombre de boucles à effectuer.

La seconde opération la plus répandue sur les blocs de mémoire, après leur initialisation, est probablement la copie. La fonction **`memcpy()`** permet de copier des blocs de mémoire disjoints. Son prototype est :

```
void * memcpy (void * destination, void * origine, size_t longueur);
```

1. Bien qu'il y ait une différence conceptuelle entre `char * chaine` et `char tableau[]`, cette distinction ne nous concernera pas ici. Pour plus de détails, on pourra se reporter à la Faq Usenet `comp.lang.c`.

Elle renvoie le pointeur sur la destination, après y avoir recopié la longueur désirée de la chaîne originale.

Attention

`memcpy()` ne peut travailler que sur des blocs de mémoire disjoints. Si les deux blocs risquent de se recouvrir, il faut utiliser la fonction `memmove()` décrite plus bas. La norme Iso-C99 a d'ailleurs ajouté au langage C un mot-clé `restrict` destiné à indiquer que les chaînes ne doivent pas se chevaucher.

La fonction `memcpy()` est par exemple très utile pour recopier tous les champs d'une structure :

```
void ma_fonction (struct ma_structure * originale)
{
    struct ma_structure copie_de_travail;
    memcpy(& copie_de_travail, originale, sizeof(struct ma_structure));
    ...
}
```

Il est du ressort du programmeur de s'assurer qu'il y a suffisamment de place pour recevoir la copie du bloc original.

Il existe une extension Gnu, **`mempcpy()`**, ayant le même prototype que `memcpy()` mais renvoyant, à la place d'un pointeur sur le début de la zone destination, un pointeur sur l'octet suivant immédiatement le dernier octet écrit dans la zone destination. Cette adresse est donc à nouveau utilisable pour une copie. Cela permet notamment de concaténer des objets de tailles différentes (en vue d'une écriture groupée sur le disque par exemple) :

```
    void *
assemble_blocs (int nb_blocs, size_t taille_bloc[], void * bloc[])
{
    void * retour;
    void * cible;
    int    taille = 0;
    int    i;

    for (i = 0; i < nb_blocs; i++)
        taille += taille_bloc[i];
    if ((retour = malloc(taille)) == NULL)
        return NULL;
    cible = retour;
    for (i = 0; i < nb_blocs; i ++)
        cible = mempcpy(cible, bloc[i], taille_bloc[i]);
    return retour;
}
```

La fonction **`memccpy()`** permet d'effectuer une copie jusqu'à la longueur désirée, ou jusqu'à avoir rencontré un caractère donné dans le bloc original. Son prototype est :

```
void * memccpy (void * destination, void * source,
                int octet, size_t longueur);
```

Si l'octet d'arrêt est trouvé dans le bloc source, il est copié dans le bloc destination, et `memccpy()` renvoie un pointeur sur le caractère suivant dans le bloc cible. Si la longueur maximale est atteinte durant la copie sans avoir rencontré l'octet d'arrêt, `memccpy()` renvoie un pointeur `NULL`.

Vous avez probablement deviné que cette routine servira avec les chaînes de caractères, pour l'implémentation de `strncpy()`. Cette fonction est aussi assez utile lorsqu'on traite des blocs de données provenant de dispositifs industriels ou scientifiques qui utilisent un octet précis de synchronisation pour délimiter le début d'un nouvel ensemble de données de longueur variable. Cette fonction permet ainsi de lire un bloc jusqu'à l'ensemble suivant, tout en s'assurant de ne pas déborder de la mémoire tampon prévue pour le traitement.

Nous avons bien précisé que les zones de mémoire ne doivent pas se chevaucher lors de l'utilisation des fonctions de type `memcpy()`. Sinon le résultat est indéfini car, durant la copie, on risque de rencontrer des octets déjà placés dans le bloc destination. À titre d'exemple naïf, supposons qu'on ait la situation suivante :

Source				Destination											
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

On espère, après avoir effectué `memcpy(destination, source, 12)`, obtenir :

Source				Destination											
A	B	C	D	A	B	C	D	E	F	G	H	I	J	K	L

Malheureusement, une fois les 4 premiers octets copiés, la routine va lire les 4 suivants, qui coïncident avec la chaîne destination, et au lieu de trouver EFGH, on retrouve ABCD :

Source				Destination											
A	B	C	D	A	B	C	D	I	J	K	L	M	N	O	P

La situation se reproduit de nouveau 4 octets plus loin :

Source				Destination											
A	B	C	D	A	B	C	D	A	B	C	D	M	N	O	P

Ce qui nous conduit finalement à ce résultat, assez éloigné de ce que nous espérions :

Source				Destination											
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D

Il est pourtant souvent nécessaire de déplacer une partie d'un bloc de données vers un emplacement le recouvrant partiellement. À titre d'exemple, on peut implémenter ainsi un buffer linéaire, où les données déjà traitées doivent être écrasées par celles qui restent à l'autre bout de la mémoire tampon. La fonction **`memmove()`** est utilisée dans ce cas. Son prototype est :

```
void * memmove (void * destination, void * source, size_t longueur);
```

Cette routine se comporte exactement comme `memcpy()` lorsque les blocs sont disjoints, mais lorsqu'ils se recouvrent, elle assure que la copie obtenue finalement sera exactement une image de la source originale au début de l'appel. Pour cela, elle compare les pointeurs source et destination, et détermine si elle doit commencer sa copie par le début (comme nous l'avons fait précédemment) ou par la fin du bloc (comme nous aurions dû le faire pour que cela fonctionne).

> **Attention**
> Le nom de cette fonction ne doit pas engendrer de confusion, il s'agit bien d'une copie, et pas d'un déplacement. Le bloc original n'est pas modifié s'il n'y a pas de recouvrement avec le bloc destination.

La fonction `memmove()` nécessite de la part de la bibliothèque C un léger surcroît de travail par rapport à `memcpy()`, puisqu'il faut qu'elle détermine le sens de progression de la copie, mais elle est quand même beaucoup plus sécurisante que cette dernière.

Il existe une fonction obsolète nommée **`bcopy()`** qui fonctionnait comme `memmove()`, mais avec le prototype suivant :

```
void bcopy (void * source, void * destination, size_t longueur);
```

Elle se comporte comme `memmove()` vis-à-vis des recouvrements de blocs, mais ne renvoie pas de pointeur. Pire, *ses arguments sont inversés* par rapport aux routines `memcpy()` et `memmove()`. Autrement dit, il ne faut plus l'utiliser !

Il est possible de comparer deux blocs de mémoire. La fonction **`memcmp()`** assure ce rôle avec le prototype suivant :

```
int memcmp (const void * bloc_1, const void * bloc_2, size_t taille);
```

Elle renvoie 0 si les deux blocs sont égaux sur la taille indiquée, sinon elle renvoie –1 ou 1 suivant le signe de la soustraction entre les premiers octets différents entre les deux blocs. Cette différence est calculée après avoir converti les octets sous forme `de int`. Voyons un exemple des divers cas possibles :

exemple_memcmp.c :

```
    #include <stdio.h>
    #include <string.h>

    void
affiche_resultats (unsigned char * bloc_1, unsigned char * bloc_2, int lg)
{
    int i;
    fprintf(stdout, "bloc_1 = ");
    for (i = 0; i < lg ; i ++)
        fprintf(stdout, "%02d ", bloc_1[i]);
    fprintf(stdout, "\n");
    fprintf(stdout, "bloc_2 = ");
    for (i = 0; i < lg ; i ++)
        fprintf(stdout, "%02d ", bloc_2[i]);
    fprintf(stdout, "\n");
```

```
    fprintf(stdout, "memcmp(bloc_1, bloc_2, %d) = %d\n",
                lg, memcmp(bloc_1, bloc_2, lg));
    fprintf(stdout, "\n");
}

    int
main (void)
{
    unsigned char bloc_1[4] = { 0x01, 0x02, 0x03, 0x04 };
    unsigned char bloc_2[4] = { 0x01, 0x02, 0x08, 0x04 };
    unsigned char bloc_3[4] = { 0x01, 0x00, 0x03, 0x04 };
    affiche_resultats(bloc_1, bloc_1, 4);
    affiche_resultats(bloc_1, bloc_2, 4);
    affiche_resultats(bloc_1, bloc_3, 4);
    return EXIT_SUCCESS;
}
```

Attention

`memcmp()` renvoie le signe du résultat de la soustraction des premiers octets différents, pas la valeur même de la différence.

```
$ ./exemple_memcmp
bloc_1 = 01 02 03 04
bloc_2 = 01 02 03 04
memcmp(bloc_1, bloc_2, 4) = 0

bloc_1 = 01 02 03 04
bloc_2 = 01 02 08 04
memcmp(bloc_1, bloc_2, 4) = -1

bloc_1 = 01 02 03 04
bloc_2 = 01 00 03 04
memcmp(bloc_1, bloc_2, 4) = 1

$
```

Il faut être très prudent avec les comparaisons de blocs de mémoire. En effet, on aurait tendance, à tort, à utiliser cette routine pour comparer des structures par exemple. Mais le compilateur insère fréquemment des octets de remplissage dans les structures ou dans les unions pour optimiser l'alignement des divers champs. Ces octets de remplissage n'ont pas de valeurs précisément définies et peuvent varier entre deux structures dont les membres sont par ailleurs égaux. On ne pourra donc pas utiliser `memcmp()` pour comparer autre chose que des données binaires « brutes » où chaque octet a une signification précise.

Comme toujours, il existe une version obsolète de cette routine provenant de BSD, **`bcmp()`**, `qui est` similaire à `memcmp()` :

```
int bcmp (const void * bloc_1, const void * bloc_2, int taille)
```

Nous nous intéresserons aux routines permettant de rechercher des sous-blocs de données au sein d'une zone de mémoire dans la section sur les recherches au cœur d'une chaîne.

Mesures, copies et comparaisons de chaînes

Avec la bibliothèque C standard, les chaînes sont représentées par une table de caractères terminée par un caractère nul permettant d'en marquer la fin. Lorsqu'on déclare une chaîne ainsi :

```
char * chaine = "Seize caractères";
```

le compilateur crée une zone de données statique initialisée, avec dix-sept caractères :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S	e	i	z	e		c	a	r	a	c	t	è	r	e	s	\0

La fonction **`strlen()`** renvoie la longueur d'une chaîne, sans compter le caractère '`\0`' final. Cette fonction est déclarée dans `<string.h>` ainsi :

```
size_t strlen (const char * chaine);
```

Pour poursuivre notre exemple, `strlen ("Seize caractères")` renvoie 16 puisque le caractère nul de fin n'est pas compté. Comme les tableaux sont accessibles en C à partir de l'élément d'indice 0, on retrouve toujours :

```
chaine [strlen (chaine)] = '\0'.
```

Lorsqu'on alloue dynamiquement la mémoire pour une chaîne, on la stocke dans un pointeur de type `char *`. Il importe à ce moment de ne pas oublier la place nécessaire pour le caractère nul final. La fonction **`strcpy()`** permet de copier une chaîne dans une autre. Il faut avoir alloué suffisamment de place dans la chaîne réceptrice. Le prototype de `strcpy()` est le suivant :

```
char * strcpy (char * destination, const char * origine);
```

La bonne méthode pour allouer la mémoire indispensable à la réception d'une copie d'une chaîne est la suivante :

```
    char * nouvelle;
    if ((nouvelle = malloc(strlen(originale) + 1)) != NULL)
        strcpy(nouvelle, originale);
    else
        perror("malloc");
```

Même une fonction aussi simple que `strlen()` peut parfois poser des problèmes. En effet, l'implémentation d'une telle routine nécessite de balayer toute la chaîne jusqu'à rencontrer un caractère nul, puis de renvoyer le nombre de caractères parcourus. La véritable implémentation est optimisée en assembleur dans la bibliothèque GlibC, mais on peut quand même en donner un équivalent fourni par Kernighan et Ritchie :

```
    size_t
strlen (const char * s)
{
    char * p = s;
    while (* p != '\0')
        p++;
    return p - s;
}
```

En fait, on renvoie ici la différence arithmétique entre le pointeur sur le caractère nul et le pointeur initial. Un problème grave peut se poser si on ne trouve pas de caractère nul. Imaginons le traitement de chaînes relativement grandes provenant d'un fichier de texte par exemple. Malheureusement, l'utilisateur s'est trompé lorsqu'il nous a fourni le nom du fichier de texte à lire et il nous a transmis un fichier constitué de données binaires, une image graphique par exemple, ne contenant – comble de malheur – aucun zéro. Que se passe-t-il alors ? La fonction `strlen()` va parcourir toute la zone de mémoire à la recherche d'un zéro et, n'en trouvant pas, va déborder sur la suite de la mémoire. Il est possible que la page suivante ne soit pas attribuée, et le programme va recevoir subitement un signal `SIGSEGV` qui va le tuer.

On peut avancer qu'il suffit, avant d'appeler `strlen()`, d'écrire de force un caractère nul à une distance arbitraire, suffisamment grande pour correspondre à la plus grande chaîne qu'on puisse traiter. Dans la plupart des cas, c'est effectivement suffisant, mais nous n'avons pas toujours un accès en écriture sur la page mémoire de la chaîne à lire, la projection d'un fichier par exemple peut avoir uniquement l'autorisation PROT_READ.

Par ailleurs, la chaîne dont il est question peut aussi être un argument d'entrée d'une routine déclarée sous forme de `const char *`, donc non modifiable (même si le compilateur ne fournit qu'un avertissement et pas une erreur). La chaîne peut aussi être une constante statique dans un segment de données protégé en écriture. Par exemple, le code suivant déclenche une erreur de segmentation `SIGSEGV` :

```
    void
modifie_chaine (char * chaine)
{
    chaine[0] = '\0';
}

    int
main (void)
{
    modifie_chaine("abc");
    return EXIT_SUCCESS;
}
```

Pour éviter de déborder d'une chaîne en recherchant sa longueur, la bibliothèque GlibC offre une fonction **`strnlen()`** qui limite la portée de la recherche de la fin de chaîne. Elle prend tout simplement un argument supplémentaire pour indiquer la longueur maximale :

```
size_t strnlen (const char chaine, size_t longueur_maxi);
```

Dans la documentation Gnu, cette fonction est indiquée comme étant équivalente à

```
strlen (chaine) < longueur_maxi ? strlen (chaine) : longueur_maxi
```

mais elle n'est heureusement pas implémentée comme cela. Tout d'abord, il faudrait stocker dans une variable le retour de `strlen()` et ne pas la rappeler deux fois. Mais de surcroît, cette fonction n'arrangerait en rien notre problème si nous attendions le retour de `strlen()` pour limiter sa valeur. En réalité, `strnlen()` est implémentée en utilisant `memchr()`, que nous verrons dans la prochaine section, et qui recherche la première occurrence du caractère nul jusqu'à une certaine distance limite.

Pour savoir si on a atteint ou non la longueur maximale, il suffit d'examiner le dernier caractère qui doit normalement être nul :

```
    taille = strnlen(chaine, TAILLE_MAXI_SEGMENT);
    if (chaine[taille] != '\0') {
        /* prévenir l'utilisateur et recommencer la saisie */
        fprintf(stderr, "La chaine fournie est trop longue \n");
        return ERREUR;
    }
```

Lorsqu'on désire obtenir une copie d'une chaîne de caractères, il n'existe pas moins de huit variantes possibles dans la GlibC. La fonction la plus courante est bien entendu **`strcpy()`** déclarée ainsi :

```
char * strcpy (char * destination, const char * source);
```

Elle copie tous les caractères contenus de la chaîne source, y compris le 0 final, dans la chaîne destination, et renvoie un pointeur sur cette dernière. Aucune protection n'est fournie en ce qui concerne les risques de débordement de la chaîne source. Pour cela, il faut utiliser **`strncpy()`** :

```
char * strncpy (char * destination, const char * source,
                size_t taille_maxi);
```

Le comportement est le suivant :

- Si la chaîne source est plus courte que la taille maximale indiquée, elle est copiée dans la chaîne destination, puis l'espace restant de la chaîne destination est rempli avec des caractères nuls jusqu'à la taille maximale. Ceci sert lorsqu'on veut comparer des zones mémoire complètes, l'état des caractères inutilisés étant fixé.
- Si la chaîne source est plus longue que la taille maximale indiquée, on ne copie que cette dernière longueur. Aucun caractère nul n'est ajouté dans la chaîne destination. Nous avons vu que cela peut être une situation à risque pour l'emploi ultérieur de `strlen()`.

Par exemple, `strncpy(destination, "ABCDEFGH", 12)` remplit la chaîne de destination ainsi :

A	B	C	D	E	F	G	H	\0	\0	\0	\0

Alors que `strncpy(destination, "ABCDEFGH", 5)` remplit la chaîne de destination ainsi

A	B	C	D	E

sans qu'il y ait de caractère nul ajouté.

Le programmeur prudent pourra donc utiliser des routines du genre :

```
    char *    destination;
    size_t    longueur;

if ((destination = malloc(LONGUEUR_MAXI_CHAINES + 1)) == NULL) {
/* traitement d'erreur */
    [...]
}
```

```
destination[LONGUEUR_MAXI_CHAINES] = '\0';
strncpy(destination, source, LONGUEUR_MAXI_CHAINES);
  [...]
longueur = strnlen(destination, LONGUEUR_MAXI_CHAINES);
if (longueur == LONGUEUR_MAXI_CHAINES) {
    /* traitement d'erreur*/
    [...]
    }
```

Les fonctions **`stpcpy()`** et **`stpncpy()`** ont exactement la même syntaxe et la même signification que `strcpy()` et `strncpy()`, mais elles renvoient un pointeur différent :

- `stpcpy()` renvoie un pointeur sur le caractère nul final de la chaîne destination.
- `stpncpy()` renvoie un pointeur sur le caractère situé dans la chaîne destination, immédiatement après le dernier caractère écrit, si la chaîne source est plus longue que la taille maximale indiquée.
- `stpncpy()` renvoie un pointeur sur le premier caractère nul écrit dans la chaîne destination si la chaîne source est plus courte que la taille maximale. La chaîne destination est dans ce cas complétée avec des zéros jusqu'à la longueur maximale, mais on renvoie un pointeur sur le premier caractère nul ajouté.

Ces fonctions sont disponibles dans la GlibC en tant qu'extensions Gnu, même s'il s'agit probablement de routines provenant du monde Dos. En renvoyant un pointeur sur la fin de la chaîne, elles permettent de faire des concaténations successives. Nous allons créer une fonction prenant en argument une chaîne destination, une longueur maximale, suivies d'un nombre quelconque de chaînes et d'un pointeur `NULL` final. Cette routine va concaténer toutes les chaînes transmises dans la chaîne destination, en surveillant qu'il n'y ait pas de débordement, caractère nul final compris.

exemple_stpncpy.c :

```
    #define _GNU_SOURCE

    #include <stdarg.h>
    #include <stdio.h>
    #include <string.h>

    void
concatenation (char * destination, size_t taille_maxi, ...)
{
    va_list arguments;
    char *  source;
    char *  retour;
    size_t  taille_chaine;

    retour = destination;
    taille_chaine = 0;
    va_start(arguments, taille_maxi);
    while (1) {
        source = va_arg(arguments, char *);
        if (source == NULL)
            /* fin des arguments */
```

```
            break;
        retour = stpncpy(retour, source, taille_maxi - taille_chaine);
        taille_chaine = retour - destination;
        if (taille_chaine == taille_maxi) {
                /* Ecraser le dernier caractère par un zéro */
                retour --;
                * retour = '\0';
                break;
        }
    }
    va_end(arguments);
}

    int
main (void)
{
     char chaine[20];

    concatenation(chaine, 20, "123", "456", "7890", "1234", NULL);
    fprintf(stdout, "%s\n", chaine);
    concatenation(chaine, 20, "1234567890", "1234567890", "123", NULL);
    fprintf(stdout, "%s\n", chaine);
    return EXIT_SUCCESS;
}
```

L'exécution nous permet de vérifier que notre concaténation fonctionne bien, tout en ne dépassant jamais la longueur maximale indiquée :

```
$ ./exemple_stpncpy
12345678901234
1234567890123456789
$
```

Il existe deux fonctions, **`strdup()`** et **`strndup()`**, particulièrement pratiques, car elles assurent l'allocation mémoire nécessaire pour stocker la chaîne destination. Elles sont déclarées ainsi :

```
char * strdup (const char *chaine);
char * strndup (const char * chaine, size_t longueur);
```

Elles renvoient toutes deux un pointeur sur la copie nouvellement allouée de la chaîne, ou `NULL` en cas d'échec dans `malloc()`. La fonction `strndup()` ne copie au plus que la longueur indiquée, *y compris* le caractère nul final. La chaîne renvoyée se termine donc toujours par un zéro. Bien entendu, il faut libérer les chaînes renvoyées en invoquant `free()` une fois qu'on a fini de les utiliser.

Deux fonctions supplémentaires existent en tant qu'extensions Gnu : **`strdupa()`** et **`strndupa()`**. Elles se présentent exactement comme `strdup()` et `strndup()`, mais la copie de chaîne est allouée dans la pile en utilisant la fonction `alloca()`, et non `malloc()`. Il ne faut donc pas tenter d'appeler `free()` avec le pointeur renvoyé, car l'espace occupé par la chaîne sera automatiquement libéré au retour de la fonction (ou lors d'un saut non local `longjmp`). Pour ces deux fonctions, on prendra donc les précautions qui s'imposent vis-à-vis de l'emploi de variables allouées dynamiquement dans la pile avec `alloca()`, comme nous l'avons vu dans le chapitre traitant de la gestion de l'espace mémoire du processus.

Aucune des fonctions de copie que nous avons examinées ne permet de copier des chaînes se recouvrant partiellement. Il est pourtant utile de déplacer des parties d'une chaîne à l'intérieur d'elle-même. Cela permet par exemple d'éliminer les espaces en début de ligne. La seule fonction acceptable pour cela est `memmove()`, mais elle nous oblige à rechercher nous-même la fin de la chaîne. Nous verrons comment implémenter de manière assez performante une élimination des blancs en début et fin de chaîne, dans la prochaine section, car nous utiliserons les fonctions `strchr()` et `strspn()` que nous analyserons alors.

Il est également fréquent d'avoir besoin d'ajouter une portion de chaîne à la fin d'une autre. Par exemple, on prépare phrase par phrase un texte en fonction de divers paramètres, puis le texte est affiché ou transmis à une routine de sauvegarde, de présentation dans un composant d'interface graphique, etc. Plusieurs méthodes sont possibles pour concaténer des chaînes, à commencer par `strcpy()` qui utilise un pointeur sur la fin de la chaîne destination. Nous avons également écrit une routine de ce type dans l'exemple précédent, avec `stpncpy()`. Il est toujours envisageable d'employer `sprintf()`. La fonction dont on se sert le plus couramment est pourtant **`strcat()`**, ainsi que son acolyte **`strncat()`** qui permet par précaution de limiter la longueur de la chaîne réceptrice. Leurs prototypes sont déclarés ainsi :

```
char * strcat (char * destination, const char * a_ajouter);
char * strncat (char * destination, const char * a_ajouter, size_t taille);
```

La taille indiquée dans l'appel de `strncat()` est celle de la portion qui peut être ajoutée à la chaîne destination. Avant l'appel de `strncat()`, la chaîne destination doit donc disposer d'une taille totale valant au moins `strlen(destination) + taille + 1` (pour le caractère nul final). L'exemple suivant va nous permettre d'utiliser `strncat()` pour concaténer les arguments d'appel de la fonction, tout en limitant la taille totale. La valeur 20 est choisie arbitrairement pour imposer une limite volontairement basse.

exemple_strncat.c :

```
    #include <stdio.h>
    #include <string.h>

    #define LG_MAXI 32 /* 20 + 12, cf. plus bas */

    int
main (int argc, char * argv[])
{
    int  i;
    int  taille;
    char chaine[LG_MAXI + 1];

    strcpy(chaine, "Arguments : "); /* déjà 12 caractères */
    for (i = 1; i < argc; i ++) {
        taille = strlen(chaine);
        strncat(chaine, argv[i], LG_MAXI - taille);
    }

    fprintf(stdout, "%s\n", chaine);
    return EXIT_SUCCESS;
}
```

Lors de l'exécution, la chaîne est bien limitée à 32 caractères (20 pour les arguments, et 12 pour l'affichage de « Arguments : »), auxquels s'ajoute un caractère nul que nous avons compté lors de la déclaration de la chaîne.

```
$ ./exemple_strncat 12345 678 90 1
Arguments : 12345678901
$ ./exemple_strncat 123456789 01 23 45678
Arguments : 123456789012345678
$ ./exemple_strncat 123456789 01 23 45678 90123
Arguments : 12345678901234567890
$
```

Ces fonctions sont loin d'être optimales pour leur rôle car elles nécessitent, au sein de la routine, de recalculer la longueur de la chaîne destination avant de commencer la copie. Le pire c'est qu'avec `strncat()` nous devons disposer, avant l'appel, de la longueur actuelle de la chaîne, et nous invoquons donc `strlen()` une fois de plus. Toutes ces étapes pourraient être évitées en utilisant les fonctions `stpcpy()` ou `stpncpy()` et en conservant la trace du pointeur renvoyé, comme nous l'avons fait dans l'exemple de `stpncpy()`.

Nous allons à présent nous intéresser aux fonctions permettant de comparer des chaînes de caractères. Ces routines peuvent bien entendu servir à des comparaisons simples, mais aussi pour trier des listes de mots par exemple. Les routines de tri que nous étudierons dans le chapitre 17 réclament en effet un pointeur sur une fonction fournissant une relation d'ordre sur l'ensemble des éléments à classer. On peut très bien utiliser les fonctions de comparaison pour implémenter cette relation d'ordre.

La routine la plus simple est **`strcmp()`** dont le prototype est :

```
int strcmp (const char * chaine_1, const char * chaine_2);
```

Si les deux chaînes sont identiques, `strcmp()` renvoie 0. Sinon, elle renvoie une valeur dont le signe correspond au résultat de la soustraction entre les premiers caractères qui diffèrent entre les deux chaînes. La comparaison des caractères est réalisée en les considérant comme des `unsigned char` (donc allant de 0 à 255), qu'on convertit en `int` pour avoir un résultat signé.

Le résultat est assez intuitif par rapport à l'ordre du dictionnaire par exemple, et correspond aux cas suivants :

- `strcmp(chaine_1, chaine_2)` = 0 si les deux chaînes sont égales.
- `strcmp(chaine_1, chaine_2)` < 0 si la première chaîne est à classer avant la seconde.
- `strcmp(chaine_1, chaine_2)` > 0 dans le cas contraire.

Voici un exemple des différents résultats possibles :

chaine 1	chaine 2	signe de `strcmp()`
ABCDE	ABCDE	strcmp() = 0
ABCDE	ABCDZ	strcmp() < 0
ABCZ	ABCDE	strcmp() > 0
ABCDE	ABC	strcmp() > 0

Le caractère de fin de chaîne étant nul, il est plus petit que tous les autres caractères. Si une chaîne est plus courte qu'une autre, elle sera donc considérée comme étant inférieure, même si tous les autres caractères sont égaux.

La fonction **`strncmp()`** fonctionne de la même manière que `strcmp()`, mais dispose d'un argument supplémentaire :

```
int strncmp (const char * chaine_1,
             const char * chaine_2,
             size_t longueur);
```

Elle ne compare que la longueur indiquée des deux chaînes, sans aller nécessairement jusqu'à leur fin. Cette fonction est particulièrement utile lorsqu'une application autorise des saisies de mots-clés abrégés. Dans ce cas, on comparera successivement la chaîne saisie avec le vocabulaire de référence, en se limitant à la longueur de la saisie. Si on trouve une correspondance, on accepte alors l'abréviation. Voici un exemple d'une telle routine, qui explore un vocabulaire contenu dans une table de chaînes de caractères. Elle renvoie le numéro du mot saisi, ou –1 en cas d'erreur. Cette fonction va même refuser les saisies ambiguës, si deux mots peuvent servir de compléments.

```
    int
recherche_correspondance (const char * saisie)
{
    int   i;
    int   longueur;
    int   trouve = -1;

    longueur = strlen(saisie);
    for (i = 0; i < Nombre_de_mots; i ++) {
        if (strncmp(chaine, Table_des_mots[i], longueur) == 0) {
            if (trouve != -1) {
                fprintf(stderr, "Saisie ambiguë, complétez le mot \n");
                return -1;
            }
            trouve = i;
        }
    }
    if (trouve == -1)
        fprintf(stderr, "Saisie inconnue \n");
    return trouve;
}
```

Le problème des fonctions `strcmp()` et `strncmp()` est qu'elles sont souvent trop rigides pour les saisies effectuées avec une interface utilisateur conviviale. On aimerait par exemple offrir à l'utilisateur la possibilité de s'affranchir de la casse[1] des caractères, c'est-à-dire des différences entre majuscules et minuscules. Même si cette différenciation est souvent importante et obligatoire (symboles du langage C, noms de fichiers sous Unix…), on peut avoir envie de relâcher la contrainte envers l'utilisateur, quitte à modifier automatiquement la saisie par la suite.

1. La casse est un terme de typographie représentant la boîte compartimentée où étaient rangés les caractères en plomb. Le haut-de-casse était occupé par les lettres majuscules, les capitales, et le bas-de-casse contenait les minuscules. Les Anglais ont conservé cette notion dans leur vocabulaire avec *uppercase* (majuscule) et *lowercase* (minuscule), que nous retrouverons dans certains noms de fonctions.

Il existe deux fonctions, **strcasecmp()** et **strncasecmp()**, avec les mêmes prototypes que strcmp() et strncmp(), et renvoyant des valeurs de retour similaires (0 pour l'égalité, et une valeur de même signe que la différence sinon). Toutefois, ces deux fonctions présentent l'avantage de ne pas être sensibles aux différences entre majuscules et minuscules. Par exemple, AbC et aBc sont considérées comme étant égales.

Mieux, ces fonctions sont directement configurables par l'utilisateur pour ce qui concerne leur comportement avec les caractères accentués. En effet, la table Ascii classique (fournie en annexe) ne contient que des caractères non accentués. Les fonctions strcasecmp() et strncasecmp() ont un comportement parfaitement normal avec ces caractères. Mais il ne s'agit là que de la première moitié de l'espace utilisable pour les valeurs d'un octet. Le standard Ascii ne normalise que les valeurs allant de 0 à 127.

Toutefois, l'utilisateur francophone sera probablement intéressé par d'autres caractères, comme é, è, à, etc. Ces derniers, ainsi que les caractères accentués utilisés par les autres langues ouest-européennes, sont regroupés par un autre standard, complémentaire, s'étendant de 128 à 255 et nommé ISO-8859-1. Il existe d'autres tables internationales pour d'autres alphabets, mais nous nous limiterons dans nos exemples au 8859-1, qui est rappelé en annexe.

Lorsqu'un utilisateur configure sa localisation, il peut indiquer, par le biais de variables d'environnement, ses préférences pour le comportement des programmes. Par exemple, les messages d'erreur des applications Gnu sont pour la plupart traduits dans la majorité des langues. Il suffit de configurer la variable d'environnement LANG ou LC_ALL pour obtenir cette traduction :

```
$ unset LC_ALL
$ unset LANG
$ ls inexistant
ls: inexistant: No such file or directory
$ export LANG=fr_FR
$ ls inexistant
ls: inexistant: Aucun fichier ou répertoire de ce type
$
```

Nous avons effacé tout d'abord les deux variables LC_ALL et LANG pour être sûr qu'il ne reste plus de localisation valide. Le premier message d'erreur de ls était en anglais. Après avoir défini LANG à fr_FR (francophone de France), le second message d'erreur est traduit. Il n'y a aucun besoin de recompiler l'application ni de modifier des fichiers système, tout se passe simplement grâce à la configuration d'une variable d'environnement. Nous consacrerons un chapitre complet à l'étude de la localisation, mais nous allons simplement indiquer dès à présent qu'en invoquant la commande suivante

```
setlocale (LC_ALL, "");
```

en début de programme, on demande aux routines de la bibliothèque C qui le peuvent de tenir compte des variables d'environnement configurées par l'utilisateur pour ses préférences.

Les routines strcasecmp() et strncasecmp() utilisent donc les règles de localisation pour déterminer si deux lettres sont égales. Il faut noter que, dans la localisation fr_FR par exemple, la majuscule associée au caractère 'é' est 'É' et pas 'E'. Dans le cas d'une saisie de données pour des comparaisons de mots-clés (moteur de recherche web ou logiciel de documentation bibliographique, par exemple), il sera probablement nécessaire d'écrire une routine personnalisée pour remplacer totalement les caractères accentués par leur correspondant sans accents.

Ceci peut se faire très facilement au moyen d'une simple table de transcodage, avec une légère complication introduite par les lettres doubles comme æ (dans *nævus* par exemple), qu'il faudra traiter comme une exception.

Voyons un exemple d'utilisation de `strcasecmp()`.

exemple_strcasecmp.c :

```
    #include <stdio.h>
    #include <string.h>
    #include <locale.h>

    int
main (int argc, char * argv[])
{
    int compar;

    setlocale(LC_ALL, "");
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s chaine_1 chaine_2\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    compar = strcasecmp(argv[1], argv[2]);
    fprintf(stdout, "%s  %c %s \n", argv[1],
                (compar > 0 ? '>' : (compar == 0 ? '=' :  '<' )),
                argv[2]);
    return EXIT_SUCCESS;
}
```

Le fait d'employer la ligne `setlocale(...)` en début de programme rend notre application sensible à la localisation :

```
$ unset LC_ALL
$ unset LANG
$ ./exemple_strcasecmp AbCd aBcD
AbCd  = aBcD
$ ./exemple_strcasecmp àÉïÔ ÀéÏô
àÉïÔ  > ÀéÏô
$ export LC_ALL=fr_FR
$ ./exemple_strcasecmp àÉïÔ ÀéÏô
àÉïÔ  = ÀéÏô
$ ./exemple_strcasecmp àÉïÔ aeio
àÉïÔ  > aeio
$
```

Dans la localisation américaine par défaut, les caractères supérieurs à 128 sont tous différents, sans lien entre eux. Dans la localisation francophone, les accentuations sont reconnues, mais le dernier exemple montre bien qu'il n'y a pas de rapprochement entre la lettre accentuée et la lettre vierge.

Le problème est que le caractère '`é`', de code ISO-8859-1 `0xE9`, est situé bien après les lettres '`a`' à '`z`' sans accents qui s'étendent de `0x61` à `0x7A`. Autrement dit, le mot *éternité* est classé après *zygomatique*. Difficile de créer un dictionnaire ainsi ! Heureusement, il existe une

fonction de comparaison prenant en compte la localisation. Cette fonction ordonne les caractères accentués à leur emplacement naturel pour la langue configurée. À titre d'exemple, voici le classement des caractères utilisés en français :

A a À à Æ æ B b C c Ç ç D d E e É é È è Ê ê Ë ë F f G g H h I i Î î Ï ï J j K k L l M m N n O o Ô ô Ö ö P p Q q R r S s T t U u Ù ù Û û Ü ü V v W w X x Y y Z z

En réalité, le classement est plus compliqué car l'ordre au sein des variantes d'une même lettre n'est pas pris en compte si la suite du mot comporte des différences. Par exemple, « tue » est placé avant « tué », mais ce dernier est situé avant « tueur ».

Ce classement est, cette fois-ci, tout à fait correct pour organiser un dictionnaire. La fonction de comparaison permettant cette organisation s'appelle **`strcoll()`** et elle a la même syntaxe que `strcmp()` :

```
int strcoll (const char * chaine_1, const char * chaine_2);
```

Il faut, bien entendu, initialiser la localisation avec `setlocale()` au début du processus. On reprend le même programme que `exemple_strcasecmp.c`, en remplaçant simplement l'appel de `strcasecmp()` par `strcoll(argv [1], argv [2])`. Voici quelques comparaisons :

```
$ unset LC_ALL
$ unset LANG
$ ./exemple_strcoll é f
é  > f
$ export LC_ALL=fr_FR
$ ./exemple_strcoll é f
é  < f
$ ./exemple_strcoll E e
E  < e
$ ./exemple_strcoll u ù
u  < ù
$ ./exemple_strcoll ù V
ù  < V
$
```

La fonction `strcoll()` est particulièrement bien adaptée pour les tris lexicographiques, en classant des données suivant l'ordre alphabétique correct. Malgré tout, elle est assez coûteuse en termes de temps, car à chaque comparaison les deux chaînes doivent être copiées dans une version modifiée pour prendre en compte la localisation. Cette modification a lieu au sein de la bibliothèque C. Lorsqu'on désire ordonner un grand nombre de chaînes, chacune d'elles est comparée à plusieurs reprises avec ses voisines et, à chaque comparaison, on repasse par l'étape de modification tenant compte de la localisation.

La bibliothèque C nous offre la possibilité d'accéder directement à la routine de modification des chaînes. Ainsi, il est possible d'obtenir une copie modifiée de chaque chaîne en fonction de la localisation. On pourra ensuite utiliser la routine `strcmp()` directement sur les chaînes modifiées, et on obtiendra le même résultat final qu'en employant `strcoll()`. Dans la localisation « C » par défaut, les chaînes copiées sont exactement identiques aux originales puisque l'ordre des caractères est celui de la table Ascii. Dans les autres localisations, les chaînes contiennent des caractères supplémentaires destinés à permettre le tri, mais rendant les copies modifiées illisibles. Il faut donc bien conserver la version originale. En fait, la modification remplace les caractères par des séquences plus ou moins longues permettant de retrouver

l'ordre naturel de tri suivant la localisation. C'est pour cela que la chaîne copiée n'est pas directement lisible. La routine de modification est **strxfrm()**, dont le prototype est le suivant :

```
size_t strxfrm (char * destination,
                const char * origine,
                size_t taille_maxi);
```

Elle copie la chaîne d'origine, en la modifiant, dans la chaîne destination, en n'y plaçant que le nombre maximal de caractères indiqué, sans compter le caractère nul final. Cette fonction renvoie le nombre de caractères nécessaires pour copier la chaîne d'origine. Lorsque la taille maximale indiquée vaut zéro, la fonction ne touche pas à la chaîne de destination. On utilise donc généralement `strxfrm()` en deux fois, le premier appel avec `strxfrm(NULL, chaine, 0)` permet de connaître le nombre de caractères nécessaires pour la destination. On effectue l'allocation (en ajoutant un octet pour le caractère nul final), et on peut appeler `strxfrm()` avec tous ses arguments à ce moment-là.

Le programme suivant démontre que l'ordre obtenu avec `strcmp()` sur des chaînes fournies par `strxfrm()` est le même que celui qui est obtenu avec `strcoll()` sur les chaînes originales.

exemple_strxfrm.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <locale.h>

    int
main (int argc, char * argv[])
{
    char * chaine_1 = NULL;
    char * chaine_2 = NULL;
    size_t taille_1;
    size_t taille_2;
    int    compar;

    setlocale(LC_ALL, "");
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s chaine_1 chaine_2\n", argv [0]);
        exit(EXIT_FAILURE);
    }
    taille_1 = strxfrm(NULL, argv[1], 0);
    taille_2 = strxfrm(NULL, argv[2], 0);
    if (((chaine_1 = malloc(taille_1 + 1)) == NULL)
     || ((chaine_2 = malloc(taille_2 + 1)) == NULL)) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    strxfrm(chaine_1, argv[1], taille_1);
    strxfrm(chaine_2, argv[2], taille_2);
    compar = strcmp(chaine_1, chaine_2);
    fprintf(stdout, "strxfrm / strcmp : %s %c %s\n", argv[1],
            (compar == 0 ? '=' : (compar < 0 ? '<' : '>')), argv[2]);
    compar = strcoll(argv[1], argv[2]);
```

```
    fprintf(stdout, "strcoll : %s %c %s\n", argv[1],
            (compar == 0 ? '=' : (compar < 0 ? '<' : '>')), argv[2]);
    return EXIT_SUCCESS;
}
```

Les comportements sont bien identiques :

```
$ ./exemple_strxfrm A a
strxfrm / strcmp : A < a
strcoll : A < a
$ ./exemple_strxfrm a à
strxfrm / strcmp : a < à
strcoll : a < à
$ ./exemple_strxfrm à B
strxfrm / strcmp : à < B
strcoll : à < B
$
```

Pour effectuer le tri d'une table de caractères, on peut créer une structure contenant un pointeur sur la chaîne originale et un pointeur sur une chaîne copie (à allouer), créer une table de ces structures et demander à une routine de tri – comme `qsort()`, que nous verrons plus loin – de faire automatiquement le classement. Il faut passer, en argument à `qsort()`, un pointeur sur une fonction de comparaison. Celle-ci utilisera `strcmp()` sur les chaînes modifiées.

exemple_strxfrm_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <locale.h>

    typedef struct {
        char    * originale;
        char    * modifiee;
    } element_t;

    int
compare_elements (const void * objet_1, const void * objet_2)
{
    element_t * elem_1 = (element_t *) objet_1;
    element_t * elem_2 = (element_t *) objet_2;
    return strcmp(elem_1 -> modifiee, elem_2 -> modifiee);
}

    void
trie_table_mots (int nb_mots, char * table_mots[])
{
    element_t * table_elements;
    size_t      taille;
    int         i;

    table_elements = calloc(nb_mots, sizeof(element_t));
    if (table_elements == NULL) {
        perror("calloc");
```

```
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < nb_mots; i ++) {
        table_elements[i].originale = table_mots[i];
        taille = strxfrm(NULL, table_elements[i].originale, 0);
        table_elements[i].modifiee = malloc(taille + 1);
        if (table_elements[i].modifiee == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }
        strxfrm(table_elements[i].modifiee,
                table_elements[i].originale,
                taille);
    }
    qsort(table_elements, nb_mots, sizeof(element_t), compare_elements);
    for (i = 0; i < nb_mots; i ++) {
        fprintf(stdout, "%s\n", table_elements[i].originale);
        free(table_elements[i].modifiee);
    }
    free(table_elements);
}

    int
main (int argc, char * argv[])
{
    setlocale(LC_ALL, "");
    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s mots...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    trie_table_mots(argc - 1, & (argv[1]));
    return EXIT_SUCCESS;
}
```

Voici un exemple :

```
$ ./exemple_strxfrm_2 exercice exécuter examiner excuse excès
examiner
excès
excuse
exécuter
exercice
$ ./exemple_strxfrm_2 exe exé
exe
exé
$ ./exemple_strxfrm_2 exerce exécute
exécute
exerce
$
```

Nous remarquons d'ailleurs au passage que l'ordre des caractères accentués par rapport aux caractères non accentués n'a d'influence sur le classement que si la suite des deux mots est différente. C'est le même comportement que dans un dictionnaire courant.

Recherches dans une zone de mémoire ou dans une chaîne

Il arrive fréquemment qu'on ait besoin de rechercher un caractère précis dans une zone de mémoire ou dans une chaîne. Cette recherche dans une zone de mémoire peut servir, par exemple, à retrouver des délimiteurs de blocs dans un ensemble de données binaires. Au sein d'une chaîne, on cherche régulièrement le caractère nul final évidemment, mais aussi des séparateurs de mots, comme l'espace ou la tabulation. Nous allons voir plusieurs fonctions permettant ce genre d'exploration.

Recherche dans un bloc de mémoire

La fonction la plus simple est **`memchr()`**, dont le prototype est le suivant :

```
void * memchr (const void * bloc, int octet, size_t longueur);
```

Elle recherche la première occurrence de l'octet indiqué en second argument, dans le bloc sur lequel on fournit un pointeur et dont on précise la longueur. Le pointeur renvoyé correspond à l'octet trouvé ou est `NULL` si aucune correspondance n'a été trouvée dans la longueur voulue.

Cela nous permet d'implémenter de manière efficace la fonction `strnlen()` que nous avions vue plus haut :

```
    size_t
strnlen (const char * chaine, size_t taille_maxi)
{
    void * fin;
    fin = memchr(chaine, 0, taille_maxi);
    if (fin == NULL)
        return(taille_maxi);
    return fin - chaine;
}
```

Précisons tout de suite que l'implémentation interne de `memchr()` dans la bibliothèque C est loin d'être triviale. Il ne s'agit pas d'un « bête » :

```
for (i = 0; i < longueur; i ++)
    if (bloc[i] == octet)
        return & (bloc[i]);
return NULL;
```

En réalité, non seulement cette routine est optimisée en assembleur, mais de plus elle emploie un algorithme astucieux permettant de faire la recherche directement dans des blocs de 4 ou 8 octets suivant la machine. Tout comme les autres fonctions d'accès à la mémoire ou aux chaînes de caractères, l'optimisation de cette routine poussera le programmeur à y avoir recours le plus souvent possible et à éviter toute implémentation personnelle d'une fonction existante.

Il existe une extension Gnu, nommée **`rawmemchr()`**, fonctionnant comme `memchr()` mais sans indiquer de longueur maximale. Étant donné l'effet dévastateur d'une telle routine quand on ne trouve pas l'octet recherché, nous nous abstiendrons de l'utiliser.

À part quelques cas précis que nous avons évoqués plus haut, la recherche de données dans un bloc de mémoire est rarement limitée à un seul octet. On a souvent besoin de déterminer la

position d'un sous-ensemble d'un bloc. Pour cela, la bibliothèque GlibC fournit une extension **memmem()** intéressante :

```
void * memmem (const char * bloc, size_t lg_bloc,
               const char * sous_bloc, size_t lg_sous_bloc);
```

Cette fonction renvoie la position de la première occurrence du sous-bloc au sein du bloc complet, ou NULL s'il n'a pas été trouvé. Il faut être très prudent avec memmem()et ne jamais lui transmettre un sous-bloc de taille nulle, car le comportement est différent suivant les implémentations de la bibliothèque C. Une attitude sage consiste à considérer le comportement de cette routine comme indéfini si le sous-bloc est vide.

Les fonctions memchr() et memmem() constituent donc les deux routines-clés pour le travail sur les blocs de mémoire. Il existe toutefois de très nombreuses autres fonctions, permettant cette fois-ci de travailler sur des chaînes de caractères.

Recherche de caractères dans une chaîne

La fonction **strchr()** est semblable dans son principe à memchr(), la limite de la recherche étant évidemment la fin de la chaîne, sans qu'on ait besoin de la préciser explicitement :

```
char * strchr (const char * chaine, int caractère);
```

Cette fonction renvoie un pointeur sur le premier caractère correspondant trouvé, ou NULL en cas d'échec. On peut rechercher n'importe quel caractère, y compris le nul final. Cela peut être intéressant dans le cas où on voudrait disposer d'un pointeur sur la fin de la chaîne, pour y ajouter quelque chose ou pour la parcourir vers l'arrière (élimination des sauts de lignes, espaces, tabulations en fin de chaîne, par exemple).

Au lieu d'écrire quelque chose comme

```
char * suite;
suite = & (origine[strlen(origine)]);
```

ou à la limite

```
suite = origine + strlen(origine);
```

qui présente les dangers de toutes les manipulations arithmétiques de pointeurs, on peut utiliser

```
suite = strchr(origine, '\0');
```

qui évite un calcul inutile.

Rappelons que dans la GlibC, les routines de recherche de caractères sont parfaitement optimisées pour parcourir la chaîne par blocs de 4 ou 8 octets, et on a tout intérêt à y faire appel plutôt que de tenter de balayer la chaîne directement. On peut employer strchr() pour rechercher des séparateurs dans des enregistrements de données se présentant sous forme de texte, comme les deux-points dans les lignes du fichier /etc/passwd par exemple, mais nous verrons un peu plus loin des fonctions mieux adaptées à ce type de travail.

La fonction **strrchr()** présente le même prototype que strchr()

```
char * strrchr (const char * chaine, int caractere);
```

mais elle s'intéresse à la dernière occurrence du caractère dans la chaîne. Elle peut servir par exemple à rechercher le dernier caractère '/' dans un chemin d'accès, pour ne conserver que le nom d'un fichier. Il existe une fonction **basename()** dans la GlibC qui effectue ce travail, mais elle n'est pas toujours définie car il y a un conflit avec une autre fonction basename() du groupe XPG. L'implémentation Gnu est en substance la suivante :

```
    char *
basename (const char * nom_de_fichier)
{
    char * retour;

    retour = strrchr(nom_de_fichier, '/');
    if (p == NULL)
        /* le nom de fichier n'a pas de préfixe */
        return nom_de_fichier;
    /*
     * On renvoie un pointeur sur le nom situé immédiatement
     * après le dernier /
     */
    return p + 1;
}
```

Il existe deux fonctions obsolètes, **index()** et **rindex()**, qui sont respectivement des synonymes exacts de strchr() et strrchr(). On risque toujours de les rencontrer dans d'anciens fichiers source, mais il ne faut plus les employer car non seulement elles sont amenées à disparaître, mais pire, les noms de ces fonctions sont mal choisis et peu révélateurs de leur rôle.

Recherche de sous-chaînes

À l'instar de memchr() qui est souvent moins utile que memmem(), les fonctions strchr() et strrchr() ont besoin d'être complétées par une routine de recherche de sous-chaîne entière. Il existe plusieurs variantes, la plus courante étant, on s'en doute, appelée **strstr()** :

```
char * strstr (const char *chaine, const char * sous_chaine);
```

Cette fonction retourne un pointeur sur la première occurrence de la sous-chaîne recherchée au sein de la chaîne mentionnée. Si aucune correspondance n'est trouvée, cette routine renvoie un pointeur NULL. Si la sous-chaîne est vide, le pointeur renvoyé correspond au début de la chaîne. Toutefois, si on désire assurer la portabilité d'un programme, on évitera ce comportement extrême, comme avec memmem(), car d'autres bibliothèques C peuvent avoir un résultat différent.

L'utilisation de strstr() est simple :

exemple_strstr.c :

```
    #include <stdio.h>
    #include <string.h>

    int
main (int argc, char * argv[])
```

```
{
    int     i;
    char *  chaine;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s chaine sous-chaine \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (strlen(argv[2]) == '\0') {
        /* Cela peut arriver si on a lancé le programme avec
         * argument "" sur la ligne de commande.
         */
        fprintf(stderr, "La sous-chaine recherchée est vide !\n");
        exit(EXIT_FAILURE);
    }
    i = 0;
    chaine = argv[1];
    while (1) {
        chaine = strstr(chaine, argv[2]);
        if (chaine == NULL)
            break;
        /* on saute la sous-chaine trouvée */
        chaine += strlen(argv[2]);
        i ++;
    }
    if (i == 0)
        fprintf(stdout, "%s ne se trouve pas dans %s\n",
                        argv[2], argv[1]);
    else
        fprintf(stdout, "%s a été trouvée %d fois dans %s\n",
                        argv[2], i, argv[1]);
    return EXIT_SUCCESS;
}
```

Voici quelques exemples d'exécution :

```
$ ./exemple_strstr abcdabcdefgabc abc
abc a été trouvée 3 fois dans abcdabcdefgabc
$ ./exemple_strstr abcdabcdefgabc abcd
abcd a été trouvée 2 fois dans abcdabcdefgabc
$ ./exemple_strstr abcdabcdefgabc abcde
abcde a été trouvée 1 fois dans abcdabcdefgabc
$ ./exemple_strstr abcdabcdefgabc abcdf
abcdf ne se trouve pas dans abcdabcdefgabc
$
```

Il existe également une extension Gnu, nommée **`strcasestr()`**, dont le fonctionnement est le même que celui de `strstr()` mais qui ne fait pas de distinction entre minuscules et majuscules. Elle est également sensible à la localisation. Pour créer le programme `exemple_strcasestr.c`, on recopie le programme `exemple_strstr.c`, en ajoutant une définition `_GNU_SOURCE` avant les inclusions d'en-têtes, pour accéder aux extensions Gnu. On insère également une

ligne `setlocale()` pour tenir compte de la localisation et, bien entendu, on remplace `strstr()` par `strcasestr()`. Voici l'exécution qui en résulte :

```
$ ./exemple_strcasestr AbcaBcABC abc
abc a été trouvée 3 fois dans AbcaBcABC
$ ./exemple_strcasestr AbcaBcABC àbc
àbc ne se trouve pas dans AbcaBcABC
$ ./exemple_strcasestr ÀéàÉ àé
àé a été trouvée 2 fois dans ÀéàÉ
$
```

Une autre variante des fonctions de recherche consiste à s'occuper des caractères appartenant à un ensemble donné. Par exemple, la fonction **`strspn()`**, dont le prototype est le suivant

```
size_t strspn (const char * chaine, const char * ensemble);
```

renvoie la longueur de la sous-chaîne initiale constituée uniquement de caractères compris dans l'ensemble fourni en argument. On peut utiliser cette routine pour éliminer les caractères blancs en début de ligne :

```
    void
elimine_blancs_en_tete (char * chaine)
{
    size_t debut;
    size_t longueur;

    debut = strspn(chaine, " \t\n\r");
    if (debut != 0) {
        longueur = strlen(chaine + debut);
        memmove(chaine, chaine + debut, longueur + 1);
        /* longueur + 1 pour avoir le caractère nul final */
    }
}
```

L'ordre des caractères dans l'ensemble n'a pas d'importance. Il existe une fonction inverse, **`strcspn()`**, renvoyant la longueur du segment initial ne contenant aucun caractère de l'ensemble transmis. Son prototype est équivalent à `strspn()` :

```
size_t strcspn (const char * chaine, const char * ensemble);
```

Il en existe également une variante, **`strpbrk()`**, qui retourne un pointeur sur le premier caractère appartenant à l'ensemble :

```
char * strpbrk (const char * chaine, const char * ensemble);
```

Lorsque cette fonction ne trouve pas de caractère contenu dans l'ensemble indiqué, elle renvoie `NULL`. Son implémentation pourrait être :

```
    char *
strpbrk (const char * chaine, const char * ensemble)
{
    size_t longueur;
    longueur = strcspn(chaine, ensemble);
    if (chaine[longueur] == '\0')
        return NULL;
```

```
    return chaine + longueur + 1;
}
```

On peut utiliser cette routine pour éliminer les sauts de ligne et retours chariot en fin de chaîne, mais également pour ignorer tous les commentaires se trouvant à la suite d'un caractère particulier, comme '#' ou '%' :

```
    void
elimine_commentaires_et_sauts_de_ligne (char * chaine)
{
    char * rejet;
    rejet = strpbrk(chaine, "\n\r#%");
    if (rejet != NULL)
        rejet[0] = '\0';
}
```

Analyse lexicale

Un programme peut parfois avoir besoin d'implémenter un *petit* analyseur lexical. Nous insistons sur le mot petit car, dès que la complexité d'un tel analyseur augmente, on a intérêt à se tourner vers des outils spécialisés, comme `lex` et `yacc`, dont les versions Gnu sont nommées `flex` et `bison`. Pour des décompositions lexicales simples, la bibliothèque GlibC offre donc une fonction nommée **`strtok()`**. Le terme *token*, qui signifie *jeton* en anglais, est le terme consacré pour désigner des éléments d'analyse lexicale (par exemple les mots-clés, mais aussi les caractères de synchronisation comme « `;` » en langage C).

La fonction `strtok()` est déclarée avec le prototype suivant :

```
char * strtok (char * chaine, const char * separateurs);
```

On passe en premier argument un pointeur sur la chaîne à analyser, mais uniquement lors du premier appel. Ce pointeur est mémorisé par `strtok()` dans une variable statique. Lorsqu'on rappellera ensuite cette fonction, on lui transmettra un premier argument `NULL`, à moins de vouloir analyser une nouvelle chaîne.

Le second argument est une chaîne de caractères contenant ce qu'on considère comme des séparateurs. Pour extraire les mots d'une phrase, on pourra ainsi employer une chaîne de séparateurs comme « `\t,;:!?-` ».

Lors de l'appel à `strtok()`, cette fonction *modifie* la chaîne transmise à l'origine en premier argument. Cette chaîne ne doit donc pas être une constante ni une variable statique susceptible d'être modifiée par d'autres fonctions de la bibliothèque. Dans de telles situations, il convient d'allouer une copie de la chaîne, avec `strdup()` ou `strdupa()` par exemple, qu'on transmettra à `strtok()`.

La fonction `strtok()` renvoie un pointeur sur le premier élément lexical, après avoir éliminé les éventuels séparateurs en début de chaîne. Lors de l'appel suivant, `strtok()` renvoie un pointeur sur le second élément lexical, et ainsi de suite jusqu'à la fin de la chaîne, où elle renvoie `NULL`.

En fait, le fonctionnement de `strtok()` est relativement simple. Elle dispose d'une variable statique initialement nulle où elle stocke le pointeur sur le début de la chaîne. Lors d'une invocation, `strtok()` recherche le premier caractère n'appartenant pas à l'ensemble des séparateurs en utilisant `strspn()`. Elle mémorise ce pointeur, car ce sera la valeur qu'elle renverra.

Ensuite elle recherche, en appelant `strpbrk()`, le premier caractère qui soit un séparateur – donc le caractère suivant la fin du mot –, puis elle le remplace par un '`\0`' et stocke le pointeur sur l'octet suivant pour reprendre son travail lors de sa future invocation.

Nous allons écrire un programme simple qui analyse les champs des lignes transmises sur son entrée standard, en utilisant les caractères blancs comme séparateurs.

exemple_strtok.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    #define LG_MAXI 256

    int
main (void)
{
    char * ligne;
    char * champs;
    int    l, c;

    if ((ligne = malloc(LG_MAXI)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    l = 1; /* l = un */
    while (fgets(ligne, LG_MAXI, stdin) != NULL) {
        fprintf(stdout, "Ligne %d\n", l);
        c = 1;
        champs = strtok(ligne, " \t");
        while (champs != NULL) {
            fprintf(stdout, "  champs %d : %s\n", c, champs);
            champs = strtok(NULL, " \t");
            c ++;
        }
        l ++;
    }
    return EXIT_SUCCESS;
}
```

Nous pouvons utiliser ce programme pour analyser le fichier `/etc/fstab` par exemple, où les champs sont séparés par des tabulations ou des espaces :

```
$ ./exemple_strtok < /etc/fstab
Ligne 1
  champs 1 : /dev/hda5
  champs 2 : /
  champs 3 : ext2
  champs 4 : defaults
  champs 5 : 1
  champs 6 : 1
```

```
Ligne 2
  champs 1 : /dev/hda6
  champs 2 : swap
  champs 3 : swap
  champs 4 : defaults
  champs 5 : 0
  champs 6 : 0

  [...]

Ligne 9
  champs 1 : none
  champs 2 : /proc
  champs 3 : proc
  champs 4 : defaults
  champs 5 : 0
  champs 6 : 0

$
```

Le fait que `strtok()` garde une variable statique globale entre deux appels le rend non réentrant. En d'autres termes, cette fonction ne doit pas être utilisée au sein d'un gestionnaire de signaux et doit être évitée dans le cadre d'un programme multithread. Pour pallier ce problème, la bibliothèque GlibC fournit deux fonctions supplémentaires où le pointeur doit être transmis en argument à chaque appel.

La première fonction, **`strtok_r()`**, est calquée sur `strtok()` avec juste un argument supplémentaire lui permettant d'être réentrante :

```
char * strtok_r (char * chaine,
                 const char * separateurs,
                 char ** pointeur);
```

Son fonctionnement est exactement identique à celui de `strtok()`, mais il faut donc lui fournir un pointeur supplémentaire, dont on n'a toutefois pas besoin de se soucier spécialement. Dans notre exemple précédent, il suffisait de modifier le programme en ajoutant une variable

```
char * pointeur;
```

et d'utiliser les appels

```
champs = strtok_r (ligne, " \t", & pointeur);
champs = strtok (NULL, " \t", & pointeur);
```

Le programme `exemple_strtok_r` fonctionne alors exactement comme `exemple_strtok`.

La seconde fonction est **`strsep()`**, qui vient de l'univers BSD. Son prototype est le suivant :

```
char * strsep (char ** pointeur, const char * separateur);
```

Globalement, elle fonctionne comme `strtok()`, mais il est du ressort du programmeur d'initialiser le pointeur fourni en premier argument pour qu'il soit dirigé vers la chaîne à traiter.

Cette routine se comporte toutefois différemment lorsqu'elle rencontre plusieurs séparateurs successivement, puisqu'elle renvoie à ce moment-là une chaîne vide alors que `strtok()`

sautait les occurrences successives de séparateurs. Il faut donc ajouter un test supplémentaire à notre programme, dont la boucle principale devient :

exemple_strsep.c :

```
    ...
    /* identique à strtok_r.c */
    ...
    while (fgets(ligne, LG_MAXI, stdin) != NULL) {
        fprintf(stdout, "Ligne %d\n", l);
        c = 1;
        pointeur = ligne;
        while (1) {
            champs = strsep(& pointeur, " \t");
            if (champs == NULL)
                break;
            if (champs[0] == '\0')
                continue;
            fprintf(stdout, "  champs %d : %s\n", c, champs);
            c ++;
        }
        l ++;
    }

    return EXIT_SUCCESS;
}
```

L'exécution présente bien entendu les mêmes résultats.

Conclusion

Nous achevons ainsi ce chapitre consacré à la gestion des chaînes de caractères et des blocs de mémoire. Nous y avons étudié en détail les routines classiques de traitement des chaînes de caractères.

Nous avons ainsi vu quelques possibilités d'analyses lexicales simples. Pour construire un véritable analyseur complet, on se penchera plutôt sur des outils spécialisés comme `flex` et `bison`, dont on trouvera une description détaillée dans [LEVINE 1994] *lex & yacc*.

Le prochain chapitre présentera des traitements plus complexes, comme les expressions régulières, ou le cryptage de données.

16

Routines avancées de traitement des blocs mémoire

Nous avons déjà observé dans le chapitre précédent un grand nombre de routines permettant d'accomplir les tâches les plus courantes du traitement de blocs ou de chaînes de caractères.

Nous allons analyser ici deux types de traitements plus rares, mais également précieux : les expressions rationnelles, qui permettent de rendre une manipulation de chaînes beaucoup plus généraliste, et les techniques de cryptage plus ou moins élaborées des blocs de données.

Utilisation des expressions rationnelles

Ce qu'on appelle expression rationnelle (*regular expression* en anglais, parfois traduit par expression régulière) est en fait un motif contenant par exemple des caractères génériques (comme '`*`' ou '`?`' dans les commandes du shell), qu'on peut mettre en correspondance avec des chaînes de caractères précises. La syntaxe des expressions rationnelles peut être très compliquée, en gérant des répétitions, des OU logiques, etc.

La bibliothèque C nous offre des fonctions permettant de vérifier si une chaîne donnée correspond à un motif ou non. Les applications de ce principe sont nombreuses, de la recherche de noms de fichiers (`/usr/include/*.h`) à l'extraction d'une chaîne particulière dans un fichier de texte (comme avec `grep`).

Une bonne partie des applications courantes conservent, sous une forme ou une autre, une liste d'objets qu'elles manipulent. Ces objets sont souvent étiquetés à destination de l'utilisateur. Offrir à celui-ci la possibilité d'afficher, de sélectionner et de rechercher tous les objets dont le nom correspond à un motif donné peut améliorer sensiblement les performances d'une application.

Nous traiterons des fonctions permettant spécifiquement de rechercher les fichiers dont le nom correspond à un certain motif dans le chapitre consacré aux accès aux répertoires.

Les fonctions génériques de traitement des expressions rationnelles sont déclarées dans le fichier `<regex.h>`. Certaines de ces fonctions sont définies par SUSv3 (ayant été introduite auparavant par Posix.2), d'autres sont bien plus anciennes et spécifiques aux applications Gnu. Si on désire utiliser uniquement les fonctionnalités portables, il suffit de définir la constante `_POSIX_SOURCE` avant l'inclusion.

Il est difficile de donner une définition des expressions rationnelles sans entrer dans une description formelle et rébarbative. Aussi, nous laisserons le lecteur se reporter à la page de manuel `regex(7)` qui, à défaut d'être un modèle de clarté, présente l'avantage d'une exhaustivité quasi totale. On peut aussi examiner la documentation de l'utilitaire `grep`, qui est probablement le programme le plus populaire pour manipuler les expressions rationnelles.

Heureusement pour nous, le programmeur n'a aucunement besoin de connaître en détail la syntaxe des expressions rationnelles, puisque justement la bibliothèque C nous offre une interface avec ce format. Seul l'utilisateur final devra se pencher sur les arcanes de ces expressions. En fait, le programmeur devra s'y intéresser un minimum, ne serait-ce que pour rédiger la documentation de son application, mais nous échapperons à la description détaillée et formelle des expressions rationnelles. En fait, nous allons à la fin de ce paragraphe fournir un programme généraliste détaillant chaque option des routines à utiliser, mais sans avoir besoin de décrire précisément les mécanismes syntaxiques mis en œuvre.

Le principe adopté pour mettre en correspondance une chaîne avec un motif donné consiste en une première étape de compilation de l'expression rationnelle. Cette compilation permet de créer une représentation interne de l'expression afin de rendre possible une comparaison rapide par la suite. Le détail de la compilation n'est pas spécifié, il s'agit d'un choix d'implémentation de la bibliothèque C. La fonction de compilation est **`regcomp()`**, dont le prototype est :

```
int regcomp (regex_t * motif_compile, const char * motif, int attributs);
```

Cette fonction prend en deuxième argument une chaîne de caractères contenant le motif à compiler, et remplit une structure de données opaque, de type `regex_t`, qu'on passe en premier argument. On pourra ensuite utiliser le motif compilé représenté par la structure de type `regex_t` pour vérifier rapidement la correspondance avec une chaîne donnée.

Le troisième argument peut contenir un ou plusieurs attributs, représentés par des constantes symboliques qu'on associe avec un OU binaire :

Constante	Signification
`REG_EXTENDED`	Le motif doit être considéré comme une expression rationnelle au format étendu. Ceci correspond à l'option `-E` de `grep`. Dans les expressions rationnelles étendues, les caractères `?`, `+`, `{`, `|`, `(`, et `)` ont une signification spéciale, alors que dans les expressions simples, il faut les préfixer avec « \ » pour obtenir le même comportement.
`REG_ICASE`	Ignorer les différences entre minuscules et majuscules lors de la mise en correspondance.
`REG_NOSUB`	On ne désire pas conserver le contenu des sous-expressions mises en correspondance. Dans ce cas, on s'intéresse uniquement à la correspondance ou non d'un motif avec une chaîne, sans avoir besoin de savoir comment les sous-expressions sont remplies. Nous détaillerons ce mécanisme un peu plus loin.
`REG_NEWLINE`	Le caractère de saut de ligne rencontré dans une chaîne ne sera pas considéré comme un caractère ordinaire, mais prendra sa signification normale. En conséquence, les caractères spéciaux `$` et `^` contenus dans un motif pourront être mis en correspondance respectivement avec les parties suivant et précédant le saut de ligne. Le caractère « `.` » ne peut plus correspondre au saut de ligne.

Lorsque la compilation réussit, `regcomp()` renvoie 0. Sinon elle renvoie une valeur d'erreur qu'on peut transmettre à la fonction **`regerror()`** dont le prototype est le suivant :

```
size_t regerror (int erreur, regex_t * motif_compile,
                 char * libelle, size_t taille_maxi);
```

Cette fonction analyse le code d'erreur passé en premier argument, ainsi que le pointeur sur le motif compilé (ou plutôt sur le motif dont la compilation a échoué) rempli par `regcomp()`. Elle en déduit un message d'erreur – malheureusement ne prenant pas encore en compte la localisation – dont elle copie, dans la chaîne passée en troisième argument, le nombre d'octets indiqué en dernier argument, caractère nul final compris. Si le message d'erreur n'a pas pu être copié en entier, il est tronqué. La fonction `regerror()` renvoie le nombre d'octets nécessaires pour stocker le message d'erreur, caractère nul compris. Il est donc possible de l'invoquer en deux passes, la première pour déterminer la longueur à allouer avec un libellé valant `NULL` et une taille maximale à zéro, la seconde pour remplir le message.

Lorsqu'on n'a pas précisé l'option `REG_NOSUB`, la bibliothèque C nous fournit des détails sur les correspondances effectuées, sans se contenter de nous dire si les chaînes concordent. Ces informations sont stockées dans des structures de type `regmatch_t`, qu'il faut allouer avant la vérification. Au sein de ces structures, deux champs nous permettent de savoir quelle portion de la chaîne correspond à chaque sous-expression.

Le nombre de sous-expressions détectées est fourni dans le champ `re_nsub` du motif compilé, de type `regex_t`, après la réussite de `regcomp()`. Toutefois, il faut allouer un élément de plus, car la fonction de comparaison nous indique aussi la portion de chaîne correspondant à l'expression complète.

Une fois que la compilation est terminée, qu'on a alloué éventuellement un tableau de structures `regmatch_t` de la taille indiquée par le champ `re_nsub+1`, on peut appeler la fonction de comparaison **`regexec()`**. Celle-ci a le prototype suivant :

```
int regexec (regex_t * motif_compile, char * chaine,
             size_t nb_sous_expr, regmatch_t sous_expr [], int attribut);
```

Cette fonction compare la chaîne et le motif compilé, et renvoie zéro s'ils concordent. Sinon, elle renvoie une valeur pouvant être :

- `REG_NOMATCH` : pas de correspondance.
- `REG_ESPACE` : pas assez de mémoire pour traiter l'expression compilée. Ceci peut se produire à cause de récurrence dans les sous-expressions. Ce cas est très rare et doit quasiment être considéré comme une erreur fatale.

Lorsque la mise en correspondance réussit, `regexec()`remplit `nb_sous_expr` éléments du tableau `sous_expr[]` avec les informations permettant de savoir quelles portions de la chaîne correspondent aux sous-expressions entre parenthèses du motif.

Les éléments du tableau `sous_expr[]`sont de structures `regmatch_t`, possédant deux champs qui nous intéressent :

- `rm_so` correspond à la position du premier caractère de la portion de chaîne mis en correspondance.
- `rm_eo` correspond à la position de la fin de la portion de chaîne mis en correspondance.

Le premier élément (d'indice 0) dans le tableau `sous_expr[]` correspond en fait à la portion équivalant à l'expression complète. Les éléments suivants concernent les sous-expressions successives.

Le dernier argument de `regexec()` contient un attribut constitué d'un OU binaire entre les constantes suivantes :

- `REG_NOTBOL` : ne pas considérer le début de la chaîne comme un début de ligne. Le caractère spécial `$` ne s'appliquera donc pas à cet endroit.
- `REG_NOTEOL` : ne pas considérer la fin de la chaîne comme une fin de ligne. Le caractère spécial `^` ne s'y appliquera donc pas.

Enfin, une fois qu'on a terminé de traiter une expression rationnelle, il faut bien entendu libérer la table des sous-expressions qu'on a allouée, mais il faut également invoquer la fonction **`regfree()`** en lui passant en argument le pointeur sur le motif compilé. Cela permet à la bibliothèque de libérer toutes les données qu'elle a allouées dans cette structure lors de la compilation. Bien sûr, ces libérations ne sont importantes que si on souhaite à nouveau compiler une autre expression régulière, mais c'est quand même une bonne habitude à prendre pour éviter les fuites de mémoire.

Nous allons écrire un programme qui prend en argument une expression rationnelle, et qui tente de la mettre en correspondance avec les lignes qu'il lira successivement sur son entrée standard. De plus, ce programme acceptera un certain nombre d'options, qui seront transmises dans les attributs des fonctions `regcomp()` et `regexec()`. Ces options sont :

Option	Argument équivalent	Fonction concernée
-e	REG_EXTENDED	regcomp()
-i	REG_ICASE	regcomp()
-s	REG_NOSUB	regcomp()
-n	REG_NEWLINE	regcomp()
-d	REG_NOTBOL	regexec()
-f	REG_NOTEOL	regexec()

Lorsque la correspondance réussit, le programme affiche les expressions et sous-expressions reconnues. Nous traitons toutes les fonctions décrites ci-dessus.

exemple_regcomp.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <regex.h>

    void
affiche_syntaxe (char * nom_prog)
{
    fprintf(stderr, "Syntaxe : %s [options] motif\n", nom_prog);
    fprintf(stderr, " Options :\n");
```

```
    fprintf(stderr, "  -e : expressions rationnelles étendues \n");
    fprintf(stderr, "  -i : pas de différences majuscule/minuscule \n");
    fprintf(stderr, "  -s : ne pas mémoriser les sous-expressions \n");
    fprintf(stderr, "  -n : gérer les sauts de lignes \n");
    fprintf(stderr, "  -d : début de chaîne sans saut de ligne \n");
    fprintf(stderr, "  -f : fin de chaîne sans saut de ligne \n");
}

    #define LG_MAXI 256

    int
main (int argc, char * argv [])
{
    int          option;
    char *       liste_options = "eisndf";
    int          option_regcomp = 0;
    int          option_regexec = 0;
    regex_t      motif_compile;
    int          erreur;
    char *       message_erreur;
    size_t       lg_message;
    size_t       nb_sous_chaines = 0;
    regmatch_t * sous_chaines = NULL;
    char         ligne[LG_MAXI];
    char         sous_chaine[LG_MAXI];
    size_t       lg_sous_chaine;
    int          i;

    opterr = 0; /* pas de message d'erreur de getopt() */
    while ((option = getopt(argc, argv, liste_options)) != -1 ){
        switch (option) {
            case 'e' :
                option_regcomp |= REG_EXTENDED;
                break;
            case 'i' :
                option_regcomp |= REG_ICASE;
                break;
            case 's' :
                option_regcomp |= REG_NOSUB;
                break;
            case 'n' :
                option_regcomp |= REG_NEWLINE;
                break;
            case 'd' :
                option_regexec |= REG_NOTBOL;
                break;
            case 'f' :
                option_regexec |= REG_NOTEOL;
                break;
            case '?' :
                affiche_syntaxe (argv [0]);
```

```
            exit (1);
        }
    }
    if (argc - optind != 1) {
        /* il manque le motif */
        affiche_syntaxe(argv[0]);
        exit(EXIT_FAILURE);
    }
    erreur = regcomp(& motif_compile, argv[argc - 1], option_regcomp);
    if (erreur != 0) {
        lg_message = regerror(erreur, & motif_compile, NULL, 0);
        message_erreur = malloc(lg_message);
        if (message_erreur == NULL) {
            perror("malloc");
            exit(EXIT_FAILURE);
        }
        regerror(erreur, & motif_compile, message_erreur, lg_message);
        fprintf(stderr, "%s\n", message_erreur);
        free(message_erreur);
        exit(EXIT_FAILURE);
    }
    if ((option_regcomp & REG_NOSUB) == 0) {
        nb_sous_chaines = motif_compile.re_nsub + 1;
        sous_chaines = calloc(nb_sous_chaines,
                                  sizeof (regmatch_t));
        if (sous_chaines == NULL) {
            perror("calloc");
            exit(EXIT_FAILURE);
        }
    }
    while (fgets(ligne, LG_MAXI, stdin) != NULL) {
        erreur = regexec(& motif_compile, ligne, nb_sous_chaines,
                         sous_chaines, option_regexec);
        if (erreur == REG_NOMATCH) {
            fprintf(stdout, "Pas de correspondance \n");
            continue;
        }
        if (erreur == REG_ESPACE) {
            fprintf(stderr, "Pas assez de mémoire \n");
            exit(EXIT_FAILURE);
        }
        fprintf(stdout, "Correspondance Ok\n");
        if ((option_regcomp & REG_NOSUB) != 0)
            continue;
        for (i = 0; i < nb_sous_chaines; i ++) {
            lg_sous_chaine = sous_chaines[i].rm_eo
                           - sous_chaines[i].rm_so;
            strncpy(sous_chaine,
                     ligne + sous_chaines[i].rm_so,
                     lg_sous_chaine);
            sous_chaine[lg_sous_chaine] = '\0';
```

```
            if (i == 0)
                fprintf(stdout, "expression : %s\n",
                        sous_chaine);
            else
                fprintf(stdout, "ss-expr %02d : %s\n",
                        i, sous_chaine);
        }
    }

    /* Ces libérations seraient indispensables si on voulait
     * compiler un nouveau motif
     */
    free(sous_chaines);
    sous_chaines = NULL;
    nb_sous_chaines = 0;
    regfree(& motif_compile);
    return EXIT_SUCCESS;
}
```

Voici quelques exemples d'exécution, mais nous encourageons le lecteur à expérimenter lui-même les différentes options des routines `regcomp()` et `regexec()`.

```
$ ./exemple_regcomp "a\(b*\)c\(de\)"
abcdefg
Correspondance Ok
expression : abcde
ss-expr 01 : b
ss-expr 02 : de
acdef
Correspondance Ok
expression : acde
ss-expr 01 :
ss-expr 02 : de
abbbbcdefg
Correspondance Ok
expression : abbbbcde
ss-expr 01 : bbbb
ss-expr 02 : de
acdf
Pas de correspondance
$
```

Rappelons que dans les expression rationnelles '*' signifie « zéro ou plusieurs répétitions du caractère précédent » et n'a donc pas son sens habituel avec le shell. Vérifions la non-différenciation majuscules / minuscules :

```
$ ./exemple_regcomp -i "a\(b*\)c\(de\)"
ABBBCDEF
Correspondance Ok
expression : ABBBCDE
ss-expr 01 : BBB
ss-expr 02 : DE
```

Avec l'option REG_NOSUB, on ne veut pas savoir comment la mise en correspondance se fait, mais juste avoir un résultat Vrai ou Faux :

```
$ ./exemple_regcomp -s "a\(b*\)c\(de\)"
abcdefg
Correspondance Ok
```

Voyons un message d'erreur transmis par regerror() lors d'une erreur de compilation :

```
$  ./exemple_regcomp "a\(b*\)c\(de"
Unmatched ( or \(
```

Enfin, avec l'option REG_EXTENDED, les expressions rationnelles sont étendues, ce qui signifie que les métacaractères prennent leur signification sans avoir besoin d'être précédés de '\' :

```
$ ./exemple_regcomp -e "a(b*)c(de)"
abbcdeff
Correspondance Ok
expression : abbcde
ss-expr 01 : bb
ss-expr 02 : de
$
```

Nous voyons que ces fonctions sont très puissantes puisqu'elles facilitent l'accès à des performances améliorées pour une application, sans nécessiter de développement complexe. Ces fonctionnalités sont en fait une extension naturelle des comparaisons de chaînes qu'on a pu étudier précédemment.

Il existe un équivalent BSD quasi obsolète puisqu'il utilise une zone de mémoire statique pour mémoriser le motif compilé. Cet ensemble est constitué par les routines **re_comp()** et **re_exec()**, déclarées dans <re_comp.h> :

```
char * re_comp (const char * motif);
int re_exec (const char * chaine);
```

Ces fonctions n'étant pas utilisables dans un environnement multithread par exemple, il vaut mieux les éviter dorénavant.

Cryptage de données

Pour terminer cet ensemble de chapitres traitant de la manipulation des blocs de mémoire et des chaînes, nous allons consacrer un moment aux routines permettant le cryptage plus ou moins complexe de données.

Cryptage élémentaire

Tout d'abord, notons rapidement l'existence de la fonction **strfry()** :

```
char * strfry (char * chaine);
```

Cette fonction est une extension Gnu qui utilise le générateur aléatoire rand() pour modifier la chaîne transmise et en créer un anagramme. Elle renvoie ensuite un pointeur sur cette même chaîne. L'utilité d'une telle fonction ne me saute pas vraiment aux yeux. Peut-être pour créer automatiquement des mots de passe ou des jeux de lettres ?

exemple_strfry.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <string.h>

    int
main (int argc, char * argv[])
{
    char * chaine;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s chaine \n", argv[0]);
        exit(EXIT_FAILURE);
    }

    chaine = strdup(argv[1]);
    strfry(chaine);
    fprintf(stdout, "%s\n", chaine);
    return EXIT_SUCCESS;
}
```

```
$ ./exemple_strfry linux
inxlu
$ ./exemple_strfry linux
nliux
$ ./exemple_strfry linux
uxlin
$ ./exemple_strfry linux
nlixu
$ ./exemple_strfry linux
xulni
$
```

memfrob() peut être une fonction un petit peu plus utile. Cette extension Gnu dispose du prototype suivant :

```
void * memfrob (void * bloc, size_t taille);
```

Elle parcourt le bloc indiqué et effectue un OU EXCLUSIF binaire octet par octet avec la valeur magique **42** (en hommage, je suppose, à Douglas Adams). Bien sûr, lorsqu'on repasse la fonction une seconde fois sur le bloc, on retrouve exactement les données d'origine.

L'intérêt de cette routine est de dissimuler grossièrement des blocs de texte qu'on pourrait sinon trouver dans le fichier exécutable (par exemple, les listes de mots-clés et de commentaires dans un jeu d'aventure). L'idée est finalement un peu la même que pour le codage ROT-13 dans les groupes Usenet, où on dissimule par exemple la solution d'une devinette ou des révélations sur un feuilleton pour que le lecteur fasse la démarche volontaire de décoder et lire le texte.

Cryptage simple et mots de passe

La fonction de cryptage la plus simple disponible dans la bibliothèque C se nomme **crypt()**. Elle est utilisée pour la transformation des mots de passe. Son prototype est déclaré dans <crypt.h> :

```
char * crypt (const char * mot_passe, const char * prefixe);
```

Elle prend deux chaînes de caractères : le mot de passe lui-même, et un préfixe que nous préciserons ci-dessous. Elle renvoie une chaîne de caractères, allouée de manière statique, contenant le mot de passe crypté.

Le principe des mots de passe sous Unix consiste à utiliser un algorithme non réversible, transformant la chaîne claire en une bouillie illisible, mais reflétant le mot de passe initial. Lors d'une tentative de connexion, le mot de passe saisi est lui aussi passé dans cet algorithme de cryptage et les deux bouillies sont alors comparées. Si elles sont égales, la connexion est acceptée. Cette méthode permet de ne conserver sur le système que des mots de passe déjà cryptés par l'intermédiaire d'un algorithme dont on ne connaît pas de fonction inverse.

La seule manière théorique d'attaquer le système est alors de se procurer un dictionnaire, de passer tous les mots dans la moulinette de cryptage, et de comparer les mots de passe cryptés avec chacun des résultats du dictionnaire. Cela pourrait être facilement exécutable, sans le préfixe qu'on ajoute. Ce préfixe a deux rôles. Tout d'abord, il permet de sélectionner entre deux types de cryptage, MD5 ou DES, et il sert ensuite à perturber le cryptage. On veut éviter qu'un pirate puisse une fois pour toutes chiffrer à l'avance tout le dictionnaire et comparer les résultats avec les mots de passe cryptés. L'introduction d'un préfixe occupant au minimum deux caractères alphanumériques l'obligerait à crypter au minimum 4 096 dictionnaires. En fait, de plus en plus, le préfixe contiendrait plutôt 8 caractères imprimables aléatoires, ce qui nécessiterait de préparer 64^8, c'est-à-dire 2^{48}, ou encore 200 000 milliards de dictionnaires.

De plus, sur les distributions Linux récentes, ce mécanisme est encore renforcé par l'utilisation des *shadow passwords*, grâce auxquels la liste des mots de passe cryptés n'est plus accessible à tous, mais uniquement à *root*.

Le cryptage utilisant MD5 est préférable à celui utilisant DES car il s'agit réellement d'une fonction à sens unique, ne permettant en aucun cas de retrouver le mot original à partir de la version cryptée. L'algorithme de cryptage MD5 est décrit en détail dans la RFC 1321, datant d'avril 1992. Ce document présente non seulement l'algorithme mais aussi des exemples de code d'implémentation. Pour utiliser le cryptage MD5, le préfixe à fournir doit obligatoirement commencer par les caractères « `$1$` ». Ensuite, on trouve jusqu'à 8 caractères, de préférence aléatoires, choisis dans l'ensemble constitué des chiffres '`0`' à '`9`', des lettres '`A`' à '`Z`' et '`a`' à '`z`', ainsi que des caractères '`.`' et '`/`'. On peut éventuellement ajouter un '`$`' à la fin du préfixe. Sinon, la fonction `crypt()` le rajoutera elle-même.

Pour utiliser le cryptage DES, on fournit un préfixe constitué de deux caractères seulement, pris dans l'ensemble décrit plus haut. Ce cryptage nécessite également que la bibliothèque GlibC ait été compilée avec un complément particulier. Si ce n'est pas le cas, lors de l'exécution du programme, la fonction `crypt()` renvoie une chaîne vide, et la variable globale `errno` contient le code `EOPNOTSUPP`.

La chaîne renvoyée par `crypt()` contient donc le préfixe fourni, intact, éventuellement complété d'un '`$`' pour le MD5, suivi de la « bouillie » correspondant au cryptage du mot de passe.

Lors de l'emploi de la fonction `crypt()` sur un système acceptant le mécanisme DES, il faut utiliser la bibliothèque `libcrypt.so` au moment de l'édition des liens en ajoutant l'option `-lcrypt` sur la ligne de commande du compilateur.

Notre premier exemple va consister à crypter le mot de passe et le préfixe passés en arguments sur la ligne de commande, et à afficher le résultat (tel qu'on pourrait le trouver dans un fichier `/etc/passwd` ou `/etc/shadow`).

exemple_crypt.c :

```
    #include <stdio.h>
    #include <unistd.h>
    #include <crypt.h>

    int
main (int argc, char * argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s mot_passe préfixe \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "%s\n", crypt(argv[1], argv[2]));
    exit(EXIT_FAILURE);
}
```

Nous utilisons un préfixe arbitraire, qui aurait dû normalement être choisi aléatoirement. Nous créons un cryptage MD5, puis un cryptage DES.

```
$ cc -Wall -g exemple_crypt.c -o exemple_crypt -lcrypt
$ ./exemple_crypt linux2.2 \$1\$abcdefgh\$
$1$abcdefgh$rpJWA.9lTJXFSyEm/t8OP1
$ ./exemple_crypt linux2.2 ab
ab74RL2di1GZ.
$
```

Nous protégeons du shell le caractère '`$`' en le faisant précéder d'un '`\`'.

Notre second exemple va consister à vérifier si le mot de passe transmis en premier argument correspond bien au cryptage fourni en second argument. Nous pouvons directement passer à la fonction `crypt()`le mot de passe crypté en guise de préfixe, elle ne prendra en considération que les caractères qui la concernent.

exemple_crypt_2.c :

```
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <crypt.h>

    int
main (int argc, char * argv[])
{
    char *  cryptage;
```

```
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s mot_passe bouillie \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    cryptage = crypt(argv[1], argv[2]);
    if (strcasecmp(cryptage, argv[2]) == 0)
        fprintf(stdout, "Vérification Ok\n");
    else
        fprintf(stdout, "Mauvais mot de passe \n");
    exit(EXIT_SUCCESS);
}
```

Nous allons vérifier un cryptage MD5 et un DES provenant de l'exemple précédent, puis nous modifierons le dernier caractère du mot de passe crypté afin de faire échouer la comparaison.

```
$ ./exemple_crypt_2 linux2.2 \$1\$abcdefgh\$rpJWA.9lTJXFSyEm/t8OP1
Vérification Ok
$ ./exemple_crypt_2 linux2.2 ab74RL2di1GZ.
Vérification Ok
$ ./exemple_crypt_2 linux2.2 \$1\$abcdefgh\$rpJWA.9lTJXFSyEm/t8OP2
Mauvais mot de passe
$
```

La fonction `crypt()`, utilisant une chaîne de caractères statique pour renvoyer son résultat, n'est pas utilisable dans un environnement multithread ou au sein d'un gestionnaire de signaux. Pour pallier ce problème, la GlibC offre une extension Gnu nommée **`crypt_r()`**, dont le prototype est le suivant :

```
char * crypt_r (const char * mot_de_passe, const char * prefixe,
                struct crypt_data * cryptage);
```

Le dernier argument est un pointeur sur une structure contenant suffisamment de place pour stocker le mot de passe crypté. Avant d'appeler cette routine, il faut mettre à zéro le champ `initialized` de cette structure ainsi :

```
    struct crypt_data cryptage;

    cryptage . initialized = 0;

    resultat = crypt_r(mot_passe, prefixe, & cryptage);
```

On notera bien que la fonction `crypt()` ne peut servir qu'à un cryptage de mot de passe. La fonction n'étant pas réversible, on ne peut pas récupérer les donnée initiales.

Cryptage de blocs de mémoire avec DES

La bibliothèque GlibC offre la possibilité de chiffrer des blocs de mémoire en utilisant l'algorithme DES. Ce système, mis au point par IBM dans les années soixante-dix, fonctionne sur le principe d'une clé privée. Il a été décrit dans le document FIPS 46-1, publié en 1988 par le gouvernement américain, et est équivalent à l'algorithme décrit sous le nom DEA Ansi X3.92-1981.

DES fonctionne en cryptant des blocs de données de 64 bits en utilisant une clé longue de 56 bits. Cette clé comportant des bits de parité, elle s'étend également sur une longueur totale de 64 bits.

Il n'est pas question de construire une véritable application cryptographique en utilisant ces fonctions. Tout d'abord, DES se servant d'un système de clé privée, il est nécessaire de disposer d'un canal de communication sûr pour transmettre la clé de décodage à son interlocuteur, ce qui est souvent aussi compliqué que d'envoyer tout le message de manière sécurisée. De plus, DES fonctionne avec des clés de 56 bits, et il est probable que la plupart des services secrets disposent d'ores et déjà de machines capables de décrypter un message en employant la force brute (essayer toutes les clés possibles jusqu'à obtention d'un texte lisible), et ceci dans un temps raisonnable. Ce système cryptographique repose en effet sur l'idée qu'un décodage par force brute nécessite un investissement informatique et un temps de calcul rédhibitoires. Bien entendu, la validité de ces deux paramètres est difficile à estimer, et de gros centres de calcul disposent de « casseurs de DES » exploitables.

Pour éviter ces désagréments, on emploiera dans des applications cryptographiques des bibliothèques fonctionnant avec d'autres systèmes plus sûrs (RSA par exemple). On peut aussi employer directement un logiciel spécialisé comme **PGP** (*Pretty Good Privacy*), dont la renommée n'est plus à faire, ou mieux, son homologue libre **GPG** (*Gnu Privacy Guard*), qui offre l'avantage d'avoir été développé en Allemagne, et n'est donc pas soumis aux restrictions d'utilisation hors des Etats-Unis qui compliquent tant la mise en service de PGP. Si on désire intégrer des appels à GPG ou à PGP depuis le corps d'une application (pour authentifier un message par exemple), on se reportera au document RFC 2440 qui décrit le standard *OpenPGP* à utiliser.

Tout en étant conscient de toutes les limitations de sécurité inhérentes à l'emploi de DES, nous pouvons toutefois vouloir l'employer dans une application pour, par exemple, crypter le contenu d'un fichier de données dans une application comptable, masquer l'identité des patients dans les dossiers d'un système d'aide au diagnostic médical, ou encore dissimuler le contenu d'une application d'agenda électronique.

En fait, de par sa nature de système à clé privée, DES est surtout utilisable dans des environnements où le même utilisateur cryptera et décryptera les données. Son intérêt principal réside dans le verrouillage de fichiers qui restent ainsi illisibles, même pour l'administrateur *root*. DES est peu recommandé lorsque les données doivent être transmises à un correspondant, à cause du problème posé par la communication de la clé.

Comme nous l'avons déjà expliqué, l'algorithme DES utilise une clé privée de 64 bits et chiffre un bloc de 64 bits pour produire un nouveau bloc de 64 bits. Il existe dans la bibliothèque GlibC des fonctions de bas niveau, `setkey()`, `encrypt()`, `setkey_r()` et `encrypt_r()`, dont l'utilisation est particulièrement pénible car elles manipulent les blocs de 64 bits sous forme de tables de 64 caractères, chaque caractère représentant un seul bit à la fois.

Heureusement, il existe deux fonctions de plus haut niveau qui nous simplifient le travail, `ecb_crypt()` et `cbc_crypt()`. ECB signifie *Electronic Code Book* et CBC, *Cipher Block Chaining*. Il s'agit de modes opératoires différents pour la normalisation de DES. Ces fonctions servent toutes deux à chiffrer ou à déchiffrer un bloc, mais `cbc_crypt()` assure un niveau de plus de chiffrage. Cette fonction effectue en effet un OU EXCLUSIF sur les blocs avant de les chiffrer, en changeant la valeur à chaque bloc. Il existe donc une chaîne de 8 octets supplémentaires à conserver avec les données, mais l'algorithme est beaucoup moins sensible à une cryptanalyse si plusieurs blocs originaux sont semblables.

La fonction `ecb_crypt()` est déclarée dans `<rpc/des_crypt.h>` ainsi :

```
int ecb_crypt (char * cle, char * bloc, unsigned longueur, unsigned mode)
```

La clé est transmise en premier argument sous forme de bloc de 8 octets. Les bits de parité de la clé doivent être positionnés correctement. Ceci est assuré par une fonction supplémentaire que nous verrons plus bas. La fonction chiffre ou déchiffre les blocs situés à partir de l'adresse transmise en second argument, jusqu'à la longueur indiquée. Cette longueur doit être un multiple de 8. Les blocs chiffrés remplacent les blocs originaux.

Le mode indiqué en dernier argument est constitué par un OU binaire entre les constantes symboliques suivantes :

Constante	Signification
`DES_ENCRYPT`	On désire crypter les données.
`DES_DECRYPT`	On désire décrypter les données. Bien entendu, une seule de ces deux constantes doit être indiquée.
`DES_HW`	Essayer d'utiliser un coprocesseur de chiffrement DES s'il en existe un sur la machine. Ceci peut améliorer sensiblement la vitesse du cryptage. Si aucun coprocesseur n'est disponible, le chiffrement sera fait de manière logicielle.
`DES_SW`	Ne pas utiliser de coprocesseur de cryptage, même s'il en existe un sur le système. Cette option peut servir à garantir que les données ne pourront pas être interceptées avec un coprocesseur truqué installé par un administrateur peu scrupuleux.

En retour, `ecb_crypt()` renvoie l'une des constantes symboliques suivantes :

Code	Signification
`DESERR_NONE`	Cryptage réussi.
`DESERR_NOHWDEVICE`	Cryptage réussi de manière logicielle, pas de coprocesseur disponible.
`DESERR_HWERROR`	Échec de cryptage dû au coprocesseur ou à l'absence du supplément « crypt » lors de la compilation de la bibliothèque C.
`DESERR_BADPARAM`	Échec de cryptage dû à de mauvais paramètres, notamment si la longueur indiquée n'est pas un multiple de 8.

Pour éviter d'avoir à tester plusieurs cas, la macro `DES_FAILED(int erreur)` prend une valeur non nulle si l'erreur est l'une des deux dernières constantes symboliques.

Le fonctionnement de `cbc_crypt() est` exactement le même, mais avec un argument supplémentaire :

```
int cbc_crypt (char * cle, char * bloc, unsigned longueur,
               unsigned mode, char * vecteur);
```

Le vecteur est un bloc de 8 octets qui sera associé par un OU EXCLUSIF au premier bloc avant son chiffrement. Ensuite, le premier bloc crypté est utilisé à nouveau dans un OU EXCLUSIF avec le second bloc avant son chiffrement, et ainsi de suite. Le phénomène inverse a lieu lors du décryptage des blocs. On emploie souvent un bloc composé de 8 octets choisis aléatoirement en guise de vecteur initial. Il faut alors conserver ce vecteur avec les données cryptées, afin de pouvoir les décoder. Une autre solution consiste à utiliser une valeur constante, par exemple 8 octets à zéro, mais à employer un premier bloc rempli aléatoirement. Bien entendu, ces deux méthodes sont équivalentes, mais dans la seconde, le vecteur aléatoire fait partie intégrante des données cryptées.

Nous avons indiqué que la clé de chiffrage devait disposer de bits de parité correctement positionnés. Pour cela, il existe une fonction d'aide, **des_setparity()**, dont le prototype est :

```
void des_setparity (char * cle);
```

On transmet à cette fonction la clé qui nous a été donnée par l'utilisateur par exemple, et elle s'occupe de placer comme il le faut les 8 bits de parité en fonction des 56 bits efficaces de la clé. Les parités sont représentées par les bits de poids faibles de chaque octet.

L'exemple que nous allons présenter ici utilisera la fonction `ecb_crypt()` pour ne pas compliquer inutilement le code avec la gestion du vecteur initial. Le programme que nous allons créer sera appelé de deux manières différentes, avec l'aide d'un lien symbolique. Il étudiera son argument numéro 0 pour savoir s'il a été invoqué sous le nom `des_crypte` ou `des_decrypte`, et adaptera alors l'argument `mode` de `ecb_crypt()`.

Notre programme prend n'importe quelle clé passée sur la ligne de commande, en la limitant à 8 caractères. Nous savons que `strncpy()` complétera la clé avec des zéros si elle a moins de 8 caractères. Nous ne recommandons pas l'utilisation directe de la clé fournie par l'utilisateur, car les clés saisies par un être humain restent dans le domaine des caractères imprimables et ont une entropie bien plus faible qu'un choix aléatoire dans l'espace allant de 0 à 255. Pour une application importante, il faudrait résoudre ce problème.

Nous arrondissons la taille du fichier à crypter au multiple de 8 supérieur. Puis, nous projetons ce fichier en mémoire avec `mmap()`. Nous pouvons alors appeler `ecb_crypt()` pour effectuer le codage.

exemple_ecb_crypt.c

```
    #define _GNU_SOURCE
    #include <fcntl.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <sys/mman.h>
    #include <sys/stat.h>
    #include <rpc/des_crypt.h>

    int
main (int argc, char * argv[])
{
    char *      nom_programme;
    int         fichier;
    struct stat etat_fichier;
    long        taille_fichier;
    char *      projection;
    char        cle[8];
    unsigned    mode;
    int         retour;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe %s : fichier clé \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    nom_programme = basename(argv[0]);
```

```
    if (strcasecmp(nom_programme, "des_decrypte") == 0)
        mode = DES_DECRYPT;
    else
        mode = DES_ENCRYPT;
    if ((fichier = open(argv[1], O_RDWR)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    if (stat(argv[1], & etat_fichier) != 0) {
        perror("stat");
        exit(EXIT_FAILURE);
    }
    taille_fichier = etat_fichier.st_size;
    taille_fichier = ((taille_fichier + 7) >> 3) << 3;
    projection = (char *) mmap(NULL, taille_fichier,
                    PROT_READ | PROT_WRITE, MAP_SHARED, fichier, 0);
    if (projection == (char *) MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fichier);
    strncpy(cle, argv[2], 8);
    des_setparity(cle);
    retour = ecb_crypt(cle, projection, taille_fichier, mode);
    if (DES_FAILED(retour)) {
        perror("ecb_crypt");
        exit(EXIT_FAILURE);
    }
    munmap(projection, taille_fichier);
    return EXIT_SUCCESS;
}
```

Voici l’utilisation du programme :

```
$ make
ln -sf exemple_ecb_crypt des_crypte
ln -sf exemple_ecb_crypt des_decrypte
cc -Wall -g    exemple_ecb_crypt.c   -o exemple_ecb_crypt
$ cat > fichier_a_crypter
Voici un fichier que nous
allons crypter avec la
bibliothèque DES.
  (Contrôle-D)
$ ./des_crypte fichier_a_crypter clélinux
$ cat fichier_a_crypter
Ñ=JA EW_§giß_êNx43Ó“P ö•_O*_’g ¤ÚiEpóÙp…w       ÑfØìø_Myƒ#…åhÛæ≥π”k¨i?$
$ ./des_decrypte fichier_a_crypter clélinux
$ cat fichier_a_crypter
Voici un fichier que nous
allons crypter avec la
bibliothèque DES.
$
```

Nous voyons bien que le fichier crypté était totalement illisible. Ce petit programme souffre beaucoup du manque d'efficacité dans la création de la clé, mais ce défaut mis à part, il pourrait servir de modèle pour la création d'un utilitaire permettant de dissimuler le contenu de fichiers personnels, même pour l'administrateur *root*.

Conclusion

Ce chapitre nous a permis de détailler l'emploi de routines particulièrement puissantes de la GlibC pour manipuler des expressions régulières ou crypter des données.

La cryptographie est un domaine complexe, nécessitant de solides connaissances mathématiques. On en trouvera une présentation claire dans [BECKETT 1990] *Introduction aux méthodes de la cryptologie*. Pour utiliser le logiciel PGP, en attendant de disposer d'une documentation sur GPG, on se tournera vers [GARFINKEL 1995] *PGP Pretty Good Privacy.*

Le prochain chapitre concernera l'utilisation de données structurées en mémoire à des fins de tri et de recherche rapide.

17

Tris, recherches et structuration des données

Les algorithmes mis en œuvre lors des opérations de tri ou de recherche de données peuvent être très simples ou au contraire extrêmement compliqués. La bibliothèque C nous propose plusieurs variantes entre lesquelles nous choisirons, au gré des caractéristiques de l'application.

Nous allons tout d'abord étudier les routines de comparaison entre éléments que l'utilisateur doit fournir aux routines de la bibliothèque C. Puis, nous examinerons les recherches linéaires dans une table, ainsi qu'une amélioration intéressante concernant les données autoorganisatrices.

Nous verrons par la suite comment trier rapidement une table, afin de pouvoir utiliser une recherche dichotomique plus rapide. Une section sera consacrée à l'étude des arbres binaires, qui sont une structure de données permettant des opérations de recherche rapides.

Nous analyserons enfin les fonctions que la bibliothèque GlibC met à notre disposition pour manipuler des tables de hachage, des structures particulièrement précieuses pour gérer des listes de mots ou de symboles, par exemple.

Fonctions de comparaison

L'essentiel des fonctions présentées dans ce chapitre nécessite de pouvoir comparer des données, afin de les ordonner ou simplement pour identifier l'élément recherché. Cette comparaison peut être effectuée sur plusieurs critères variant en fonction du type de données stockées. Lorsqu'il s'agit simplement de valeurs numériques, on peut très bien utiliser les comparaisons classiques <, =, ou >. Lorsqu'on doit comparer des mots, on peut imaginer employer `strcmp()`, quoique ce ne soit pas forcément le meilleur choix. Lorsqu'on veut comparer des enregistrements d'une base de données, il est nécessaire de définir un champ

comme clé de recherche pour que la comparaison entre deux enregistrements s'effectue sur leurs membres ainsi choisis.

Pour homogénéiser les opérations, les routines de tri et de recherche reçoivent en argument un pointeur sur une fonction écrite par l'utilisateur, et qui devra prendre en arguments deux éléments et renvoyer une valeur négative, nulle ou positive selon que le premier argument est considéré comme inférieur, égal ou supérieur au second.

Le prototype d'une routine de comparaison sera donc :

```
int comparaison (const void * element_1, const void * element_2);
```

Bien sûr, celle-ci devra être adaptée au type de données manipulées par le programme. Les arguments doivent être déclarés comme des pointeurs `void *`, mais la bibliothèque C l'invoque en utilisant des pointeurs sur les objets à comparer. Voici quelques exemples :

```
    int
compare_entiers (const void * arg_1, const void * arg_2)
{
    int entier_1 = * ((int *) arg_1);
    int entier_2 = * ((int *) arg_2);
    return entier_1 - entier_2;
}

    int
compare_chaines (void * arg_1, void * arg_2)
{
    return strcasecmp((const char *) arg_1, (const char *) arg_2);
}
```

Nous voyons que, dans ce dernier cas, on aurait pu prendre directement un pointeur sur la fonction `strcasecmp()`, à condition de forcer son type par un *cast* explicite.

Voici un exemple de comparaison de données structurées :

```
    typedef struct {

        char *  nom;
        char *  prenom;
        time_t  date_naiss;
        char *  lieu_naiss;

        /* Champs non pris en compte dans la comparaison */
        time_t  date_d_inscription;
        long    livre_emprunte;
    /* ... etc ... */
    } individu_t;

    int
compare_identites (const void * element_1, cont void * element_2)
{
    individu_t * individu_1 = (individu_t *) element_1;
    individu_t * individu_2 = (individu_t *) element_2;
    int          comparaison;
```

```
    comparaison = strcasecmp(individu_1->nom, individu_2->nom);
    if (comparaison != 0)
        return comparaison;

    comparaison = strcasecmp(individu_1->prenom, individu_2->prenom);
    if (comparaison != 0)
        return comparaison;

    comparaison = individu_1->date_naiss - individu_2->date_naiss;
    if (comparaison != 0)
        return comparaison;

    comparaison = strcasecmp(individu_1->lieu_naiss,
                             individu_2->lieu_naiss);
    return comparaison;
}
```

Nous faisons ici une comparaison successive sur les quatre critères d'identification. Nous prendrons donc une clé primaire représentée par le nom, puis des clés secondaires constituées successivement par le prénom, la date et le lieu de naissance. Nos enregistrements peuvent bien entendu contenir des champs qui ne sont pas pris en compte lors de la comparaison.

Remarquons au passage que, dans certains cas, l'emploi de la comparaison `strcasecmp()` n'est pas le plus approprié. Lors des recherches sur les noms de famille par exemple, il existe toujours une marge assez importante d'incertitude concernant les fautes de frappe, les inversions de lettres dans un nom épelé au téléphone (je peux en témoigner personnellement…), ou simplement des problèmes de mauvaise compréhension dus à un accent prononcé. Pour assurer une certaine tolérance, on peut utiliser un algorithme de phonétisation. Il en existe de nombreux, comme Soundex ou Métaphone, qui sont largement employés dans les logiciels de généalogie par exemple, et qui permettent de réduire un nom à ses consonnes les plus importantes. Les règles de phonétisation variant suivant la langue, il est conseillé de rechercher un algorithme adapté aux noms à comparer.

Pour simplifier les prototypes des routines de recherche et de tri, la bibliothèque GlibC définit un type spécial, nommé **`comparison_fn_t`**, sous forme d'extension Gnu, qui correspond à une fonction de comparaison. Ce type est défini, en substance, ainsi :

```
typedef int (* comparison_fn_t) (const void *, const void *);
```

Lorsque nous trouverons, dans une liste d'arguments, une déclaration

```
int fonction_de_tri (... , comparison_fn_t compare, ...);
```

cela signifiera que la fonction `compare()` est du genre :

```
int compare (void * element_1, void * element2);
```

La plupart des routines permettent de trier des tables contenant des éléments de taille constante. Le tri des données de taille variable (chaînes de caractères par exemple) ne pose pas de problème puisqu'il suffit d'ajouter un niveau d'indirection supplémentaire en triant en réalité une table de pointeur sur les données de taille variable. Bien entendu, il faudra tenir compte correctement de cette indirection dans la routine de comparaison.

Recherche linéaire, données non triées

Les premières fonctions que nous étudierons permettent de faire une recherche linéaire dans une table, aussi appelée *recherche séquentielle*. Il s'agit simplement de parcourir toute la table jusqu'à trouver l'élément correspondant à la clé recherchée. Cette méthode n'a d'intérêt que si la table n'est pas ordonnée car, dans le cas contraire, nous verrons des routines beaucoup plus rapides pour accéder aux données.

On peut s'interroger sur l'intérêt de conserver une table de données non ordonnée, alors qu'il existe des routines de tri simples et performantes. En fait, la recherche dans une table triée n'est intéressante que si le nombre d'éléments est suffisamment grand et si la table ne subit que peu de modifications. En effet, l'insertion ou la suppression de données sont obligatoirement plus coûteuses dans une table triée que dans une table non ordonnée, puisqu'il faut faire appel à des routines spécialisées pour placer l'enregistrement au bon endroit.

Si notre table est « petite » (au maximum quelques dizaines d'enregistrements), et si la routine de comparaison est simple et rapide, il est plus commode de laisser la table non ordonnée et d'utiliser une recherche séquentielle. Ce choix sera également plus judicieux si la table change beaucoup. Cela signifie qu'on renouvelle le contenu de la table en permanence, et qu'un enregistrement donné n'est pas recherché plus de deux ou trois fois durant son existence. Ainsi, j'ai employé une recherche séquentielle dans un logiciel dans lequel on reçoit en permanence des positions d'avions en approche finale sur un aéroport. Chaque enregistrement n'existant dans notre liste que pendant une durée assez courte, alors qu'il y a des ajouts et des suppressions pratiquement toutes les secondes, l'utilisation d'une liste triée ne se justifiait pas.

Les deux routines de recherche linéaire offertes par la bibliothèque GlibC sont nommées **`lfind()`** et **`lsearch()`**. Leurs prototypes sont les suivants :

```
void * lfind (const void * cle, const void * base,
              size_t * nb_elements, size_t taille,
              comparison_fn_t compare);
```

et

```
void * lsearch (const void * cle, void * base,
                size_t * nb_elements, size_t taille,
                comparison_fn_t compare);
```

Elles sont déclarées dans le fichier `<search.h>`.

La première routine recherche l'élément qui correspond à la clé fournie en premier argument dans la table commençant à l'adresse passée en second argument contenant `* nb_elements`, et chaque élément ayant la taille indiquée en quatrième position Pour chercher la donnée correspondant à la clé, la fonction de comparaison fournie en dernière position est employée. On doit passer un pointeur sur le nombre d'éléments, et non la véritable valeur, même si son contenu n'est pas modifié par `lfind()`. Si la routine trouve un élément correspondant à la clé, elle renvoie un pointeur dessus. Sinon, elle renvoie `NULL`.

Voici un exemple d'utilisation avec la structure de données `individu_t` que nous avons définie plus haut :

```
    static individu_t * table_individus = NULL;
    static size_t       nb_individus = 0;
```

```
    individu_t *
donne_individu (const char * nom, const char * prenom,
                time_t date, const char * lieu)
{
    individu_t   cle;
    individu_t * retour;

    cle . nom = nom;   /* On copie le pointeur, pas la chaîne */
    cle . prenom = prenom;
    cle . date_naiss = date;
    cle . lieu_naiss = lieu;

    retour = lfind(& cle, table_individus, & nb_individus,
                   sizeof (individu_t), compare_identites);
    if (retour != NULL)
        return retour;

    /* On ne l'a pas trouvé, on va en créer un nouveau */
    table_individus = realloc(table_individus,
                          sizeof(individu_t) * (nb_individus + 1));
    if (table_individus == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    table_individus[nb_individus].nom = strdup(nom);
    table_individus[nb_individus].prenom = strdup(prenom);
    table_individus[nb_individus].date_naiss = date;
    table_individus[nb_individus].lieu = strdup(lieu);
    time (& (table_individus[nb_individus].date_inscription));
    table_individus[nb_individus].livre_emprunte = -1;

    nb_individus ++;
    return (& (table_individus[nb_individus - 1]));
}
```

Nous avons utilisé dans cet exemple les structures déjà définies plus haut, mais on remarquera par ailleurs que la gestion d'un fichier de ce type – probablement les inscriptions dans une bibliothèque – n'est justement pas adaptée à une organisation séquentielle, puisque les données varient peu et que le nombre d'enregistrements est certainement assez conséquent.

La fonction `lsearch()` recherche également l'enregistrement de manière séquentielle dans la table fournie, mais si elle ne le trouve pas, elle ajoute un enregistrement à la fin, et incrémente l'argument `nb_elements`, sur lequel on doit passer un pointeur, comme avec `lfind()`. Cela signifie qu'il faut être sûr avant d'appeler `lsearch()` de disposer d'au moins un emplacement supplémentaire libre dans la table. On l'utilise parfois en effectuant des allocations « par blocs », afin de réduire le nombre d'appels à `realloc()`. Voici un exemple :

```
static individus * table_individus = NULL;
static size_t      nb_individus = 0;
static size_t      contenance_table = 0;

    #define  NB_BLOCS_AJOUTES  64
```

```
   individu_t *
donne_individu (const char * nom, const char * prenom,
                time_t date, const char * lieu)
{
    individu_t   cle;
    individu_t * retour;

    if (contenance_table == nb_individus) {
        contenance_table += NB_BLOCS_AJOUTES;
        table_individus = realloc(table_individus,
                      contenance_table * sizeof(individu_t));
        if (table_individus == NULL) {
            perror("realloc");
            exit(EXIT_FAILURE);
        }
    }

    cle.nom = nom;    /* On copie le pointeur, pas la chaîne */
    cle.prenom = prenom;
    cle.date_naiss = date;
    cle.lieu_naiss = lieu;

    retour = lfind(& cle, table_individus,
                & nb_individus, sizeof(individu_t),
                compare_identites);
    return retour;
}
```

On notera que, dans le cas d'une recherche séquentielle, la fonction de comparaison doit simplement renvoyer 0 si les éléments concordent, et une autre valeur sinon. On n'a pas besoin d'indiquer si la première est inférieure à la seconde ou non.

En fait, il est très commode d'appeler les routines `lfind()` ou `lsearch()` si on désire implémenter une table non triée pour débuter le développement d'une application, quitte à se tourner ensuite vers une implémentation plus structurée si le besoin s'en fait sentir. Les routines de recherche dans les tables ordonnées ou dans les arbres binaires ont une interface quasi identique, et la modification d'implémentation est facile.

Toutefois, si on désire conserver une table non triée, et si les fonctionnalités de recherche dans cette table sont critiques pour l'application, on peut envisager de réimplémenter ses propres routines à la place de celles de la bibliothèque GlibC. C'est l'un des rares cas où une réécriture de fonctions existantes peut apporter quelque chose de sensible à une application, sans risque d'erreur, vu la simplicité de l'algorithme utilisé.

La fonction `lfind()` est implémentée, en substance, dans la GlibC ainsi :

```
    void *
lfind (const void * cle, const void * base, size_t * nb_elements,
        size_t taille, comparison_fn_t compare)
{
    const void * retour = base;
    size_t       compteur = 0;
```

```
    while ((compteur < *nb_elements) && ((*compare)(cle, retour) != 0)) {
        retour += taille;
        compteur ++;
    }
    return (compteur < *nb_elements ? retour : NULL);
}
```

On peut reprocher deux choses à cette fonction :

- Elle appelle pour chaque enregistrement la routine de comparaison, alors qu'on pourrait éviter la surcharge de code due à une invocation de fonction en intégrant directement le code de comparaison dans la recherche séquentielle.
- Elle effectue deux tests à chaque itération, en vérifiant à la fois si le compteur a atteint le nombre d'éléments dans la table et si la comparaison a réussi.

En fait, pour éviter de dupliquer le test à chaque itération, il suffit d'ajouter un élément fictif à la fin de la table, dans lequel on copie la clé recherchée. On ne fait plus que la comparaison à chaque itération. Lorsqu'on sort de la boucle, on vérifie alors si on avait atteint le dernier élément ou non. Cette méthode oblige à toujours disposer d'un emplacement supplémentaire en fin de table, mais il suffit d'allouer un élément de plus à chaque appel `realloc()`.

Nous pouvons alors écrire une routine spécialisée pour nos données. Par exemple, pour rechercher un entier dans une table non triée :

```
    int *
recherche_entier (int cle, int * table, int nb_entiers)
{
    int *   resultat = table;

    /* on sait qu'on dispose d'un élément supplémentaire */
    table[nb_entiers] = cle;
    while (cle != *resultat)
        resultat += sizeof(int);
    if (resultat == & (table[nb_entiers]))
        return NULL;
    return resultat;
}
```

Ceci nous permet d'augmenter les performances de cette recherche, en l'adaptant à nos données. Une autre amélioration peut parfois être apportée en utilisant une organisation automatique des données.

La recherche séquentielle balaye tous les enregistrements jusqu'à trouver celui qui convient. Lorsque le nombre d'enregistrements croît, la durée de la recherche augmente dans la même proportion. On dit que la complexité de cet algorithme s'exprime en *O(N)*, *N* étant le nombre de données dans la base.

Lorsque tous les enregistrements présents ont la même probabilité d'être recherchés, le parcours séquentiel balaye, en moyenne, N/2 éléments. Toutefois, ceci n'est vrai que si les données sont équiprobables, c'est-à-dire si tous les enregistrements font l'objet d'une recherche le même nombre de fois.

Or, dans de très nombreuses situations, certaines données sont beaucoup plus sollicitées que d'autres. À titre d'exemple, les secteurs d'un disque dur ou les mots d'un lexique obéissent

plutôt à une loi dite 80-20, c'est-à-dire que 20 % des données font l'objet de 80 % des recherches. Et au sein de ces 20 %, la même loi peut se répéter.

Autrement dit, on a tout intérêt à trouver un moyen de placer en tête de table les données les plus fréquemment recherchées. Pour cela, il existe une méthode simple : à chaque fois qu'une recherche aboutit, l'élément retrouvé est permuté avec celui qui le précède dans la table. Les données les plus demandées vont donc remonter automatiquement au cours des recherches successives, afin de se trouver aux places de choix, celles qui nécessitent un balayage minimal. De même, les enregistrements qu'on ne réclame jamais vont descendre en fin de table, là où la recherche dure le plus longtemps.

La table s'organisant au fur et à mesure des demandes de l'utilisateur, les résultats sont parfois étonnamment bons, surtout si on remarque que la plupart des recherches successives ne sont pas indépendantes et sont dictées par un centre d'intérêt commun qui réclame parfois le même élément à plusieurs reprises. C'est un phénomène un peu similaire à celui des mémoires cache en lecture, qui permettent d'améliorer sensiblement les performances d'un disque dur.

On peut, par exemple, implémenter ce mécanisme en ajoutant à la suite d'un appel à `lfind()` :

```
element_t * retour;
element_t   echange;

retour = lfind (cle, table, & nb_elements,
                sizeof (element_t), compare_elements);
if ((retour != NULL) && (retour != table)) {
    memcpy(& echange, retour, sizeof(element_t));
    memcpy(retour, retour - sizeof(element_t), sizeof(element_t));
    memcpy(retour - sizeof(element_t), & echange, sizeof(element_t));
}
```

Bien entendu, avec des types entiers ou réels par exemple, l'échange est bien plus simple. Attention, répétons que les tables autoorganisatrices fonctionnent uniquement si les données n'ont vraiment pas la même probabilité d'être recherchées. Ceci nécessite donc d'analyser précisément les éléments dont on dispose lors de l'implémentation de l'application.

Nous avons ainsi vu les mécanismes les plus simples pour rechercher des données non organisées, ainsi que quelques astuces pouvant améliorer les performances. Malgré tout, dans la majeure partie des cas, il est préférable d'essayer de trier le contenu de notre ensemble d'informations afin d'obtenir des recherches beaucoup plus rapides.

Recherches dichotomiques dans une table ordonnée

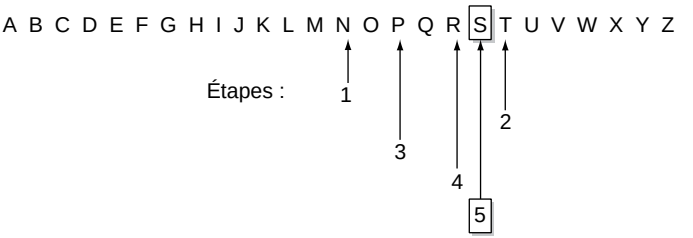
Lorsqu'on dispose d'une table où les données sont triées, une recherche est bien plus rapide. Il suffit en effet d'utiliser un algorithme de recherche dichotomique pour obtenir des performances très intéressantes. Le principe de la recherche dichotomique est simple :

- on choisit un élément au centre de la table triée et on le compare avec la clé recherchée ;
- si la clé est égale à l'élément, on a fini avec succès ;
- si la clé est plus petite que l'élément, on réitère le processus sur la moitié inférieure de la table ;
- sinon, on recommence en partant de la moitié supérieure de la table ;

- si la clé recherchée ne se trouve pas dans la table, on finira par se retrouver avec une portion réduite à un seul élément, auquel cas on finira l'algorithme en échec.

Cet algorithme nous garantit une complexité en *O(log(N))*, ce qui signifie que lorsque le nombre N de données augmente, la durée de la recherche croît proportionnellement à log(N). Or, pour des valeurs suffisamment grandes, log(N) est très inférieur à N. Cette recherche dichotomique est donc largement plus rapide qu'une recherche séquentielle.

Figure 17.1
Recherche dichotomique dans une table triée



Si la recherche dichotomique paraît simple à première vue, sa programmation l'est beaucoup moins. Cet exercice classique révèle des subtilités de mise en œuvre, et la première tentative d'implémentation est rarement exempte de défauts. Aussi, on se limitera autant que possible à utiliser directement la routine **`bsearch()`** de la bibliothèque C, qui implémente la recherche dichotomique de manière exacte et optimisée, définie dans `<stdlib.h>` :

```
void * bsearch (const void *cle, const void * table,
                size_t nb_elements, size_t taille,
                comparison_fn_t compare);
```

La seule différence avec le prototype de `lsearch()` est qu'on passe le nombre d'éléments du tableau et pas un pointeur sur ce nombre.

Cette fonction renvoie un pointeur sur l'élément recherché, ou `NULL` en cas d'échec. La fonction de comparaison doit renvoyer une valeur positive ou négative en fonction de la position des deux éléments comparés, et non plus simplement une valeur nulle ou non nulle comme dans la recherche séquentielle. Si plusieurs éléments de la table sont égaux, l'algorithme ne précise pas celui qui sera trouvé (contrairement à `lsearch()` qui rencontre toujours le premier d'abord).

Avant de pouvoir utiliser `bsearch()`, il faut ordonner les données. Pour cela, il existe de nombreux algorithmes plus ou moins performants, et la bibliothèque C en implémente deux, le choix étant effectué au moment de l'appel en fonction de la taille des données et de la disponibilité mémoire.

L'algorithme le plus efficace garantit une complexité en *O(N.log(N))*, c'est-à-dire que la durée du tri croît proportionnellement à *N.log(N)* lorsque le nombre N d'éléments augmente. Indiquons que les algorithmes de tri « évidents », comme le célèbre tri à bulles, ont une complexité en $O(N^2)$, et rappelons que lorsque N est déjà moyennement grand – quelques centaines d'éléments – log(N) est nettement inférieur à N.

Cet algorithme nécessite un espace de stockage de taille identique à la table à trier. Le principe consiste à séparer récursivement la partition à trier en deux ensembles de taille identique à un élément près. Ces deux ensembles sont alors triés séparément par appel récursif de

l'algorithme. Ensuite, on mélange les données triées dans la mémoire temporaire en les parcourant une seule fois, par comparaisons successives. La mémoire temporaire est alors recopiée dans la table à présent ordonnée.

Cet algorithme est optimal en termes de rapidité, mais il nécessite un espace de stockage temporaire qui peut parfois être exagéré si on trie par exemple le contenu d'un fichier projeté en mémoire avec `mmap()`. Aussi, la bibliothèque GlibC en propose un second qui est presque aussi efficace et qui ne nécessite pas de mémoire auxiliaire : le **quicksort** (tri rapide). Ce tri, décrit par C. Hoare en 1952, est particulièrement célèbre, et la GlibC implémente de surcroît des améliorations pour augmenter encore ses performances.

Le *quicksort* repose sur la division successive de la table à trier en partitions de taille de plus en plus réduite. Le principe consiste à choisir dans la partition à ordonner une valeur médiane, dite *pivot*, et à scinder la partition en deux sous-ensembles distincts, l'un contenant uniquement des valeurs inférieures au pivot, et l'autre comprenant seulement des valeurs supérieures au pivot. Pour effectuer ce découpage rapidement, on utilise deux pointeurs : l'un partant du bas de la partition et remontant progressivement jusqu'à rencontrer une valeur plus grande que le pivot, et l'autre partant symétriquement du haut de la partition pour descendre jusqu'à trouver un élément inférieur au pivot. Si les deux pointeurs se sont croisés, la séparation en deux sous-partitions est finie, sinon on échange les deux éléments rencontrés et on continue. Le processus est alors répété sur les deux nouvelles partitions, jusqu'à avoir des sous-ensembles ne contenant que trois éléments ou moins, et la table originale est alors entièrement triée.

L'éventuelle complication avec le *quicksort* réside dans le choix du pivot. Dans l'algorithme original, on propose d'utiliser comme pivot le premier élément de la table, ce qui simplifie la suite des opérations puisqu'il suffit de placer le pointeur bas sur le deuxième élément et le pointeur haut sur le dernier, sans se soucier de rencontrer le pivot lui-même. Toutefois, cela pose un grave problème sur les tables déjà ordonnées. En effet, le découpage obtenu est alors catastrophique puisqu'il contient une sous-partition ne comprenant qu'un seul élément, et une seconde comportant les N – 1 autres. La complexité de l'algorithme n'est plus $O(N.log(N))$ mais approche au contraire $O(N^2)$.

Pour éviter ce problème, la GlibC choisit comme pivot une valeur médiane entre le premier élément du tableau, le dernier et un élément placé au milieu. Même si la table est déjà ordonnée, la performance du tri reste intacte. De même, la bibliothèque GlibC évite d'utiliser le *quicksort* lorsque la taille des partitions devient petite (quatre éléments en l'occurrence), et elle se tourne alors vers un tri par insertion qui est plus efficace dans ce cas. Enfin, les performances de l'implémentation sont encore améliorées en évitant d'utiliser la récursivité naturelle de l'algorithme, mais en gérant directement une liste des partitions à traiter.

La GlibC emploie donc autant que possible le tri avec une mémoire auxiliaire, sinon elle se tourne vers le *quicksort*. La routine **`qsort()`**, déclarée dans `<stdlib.h>`, qui tire son nom du *quicksort* utilisé dans l'implémentation traditionnelle sous Unix, est très simple d'utilisation :

```
void qsort (void * table,
            size_t nb_elements, size_t taille_element,
            comparison_fn_t compare);
```

Voici un exemple de programme qui crée une table de valeurs aléatoires, puis qui invoque `qsort()` pour les trier.

exemple_qsort.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
compare_entiers (const void * elem_1, const void * elem_2)
{
    return (* ((int *) elem_1) - * ((int *) elem_2));
}

    #define NB_ENTIERS 100

    int
main (void)
{
    int table_entiers[NB_ENTIERS];
    int i;

    for (i = 0; i < NB_ENTIERS; i ++) {
        /* On limite un peu la taille des entiers pour l'affichage */
        table_entiers[i] = rand() & 0xFFFF;
        fprintf(stdout, "%05d ", table_entiers[i]);
    }
    fprintf(stdout, "\n \n");
    qsort(table_entiers, NB_ENTIERS, sizeof(int), compare_entiers);
    for (i = 0; i < NB_ENTIERS; i ++)
        fprintf(stdout, "%05d ", table_entiers[i]);
    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution :

```
$ ./exemple_qsort
17767 09158 39017 18547 56401 23807 37962 22764 07977 31949 22714 55211 16882 07931
➥ 43491 57670 00124 25282 02132 10232 08987 59880 52711 17293 03958 09562 63790
➥ 29283 49715 55199 50377 01946 64358 23858 20493 55223 47665 58456 12451 55642
24869 35165 45317 41751 43096 23273 33886 43220 48555 36018 53453 57542 30363 40628
➥ 09300 34321 50190 07554 63604 34369 62753 48445 36316 61575 06768 56809 51262
➥ 54433 49729 63713 44540 09063 33342 24321 50814 10903 47594 19164 54123 30614
55183 42040 22620 20010 17132 31920 54331 01787 39474 52399 36156 36692 35308 06936
➥ 32731 42076 63746 18458 30974 47939

00124 01787 01946 02132 03958 06768 06936 07554 07931 07977 08987 09063 09158 09300
➥ 09562 10232 10903 12451 16882 17132 17293 17767 18458 18547 19164 20010 20493
➥ 22620 22714 22764 23273 23807 23858 24321 24869 25282 29283 30363 30614 30974
31920 31949 32731 33342 33886 34321 34369 35165 35308 36018 36156 36316 36692 37962
➥ 39017 39474 40628 41751 42040 42076 43096 43220 43491 44540 45317 47594 47665
➥ 47939 48445 48555 49715 49729 50190 50377 50814 51262 52399 52711 53453 54123
54331 54433 55183 55199 55211 55223 55642 56401 56809 57542 57670 58456 59880 61575
➥ 62753 63604 63713 63746 63790 64358
$
```

La routine `qsort()` de la bibliothèque GlibC étant pratiquement optimale, il est fortement conseillé d'y faire appel aussi souvent que possible, et de n'implémenter sa propre routine de tri que pour des applications vraiment spécifiques.

L'inconvénient que pose la fonction `bsearch()` est qu'il n'est pas facile d'ajouter simplement un élément si on ne le trouve pas. Pour cela, il faut insérer l'élément à la fin de la table par exemple, et invoquer `qsort()` pour la trier à nouveau. C'est intéressant si on peut grouper de multiples ajouts, mais peu efficace pour des ajouts isolés et fréquents. Voici donc une routine qui peut servir à ajouter un seul élément si on ne le trouve pas. Elle suppose que la table contient suffisamment de place pour adjoindre au moins une donnée.

```
    void *
b_insert (const void * cle, const void * table,
          size_t * nb_elements, size_t taille_element,
          int (* compare) (const void * lm1, const void * lm2))
{
    const void * element;
    int          comparaison;

    size_t       bas  = 0;
    size_t       haut = (* nb_elements);
    size_t       milieu;

    while (bas < haut) {

        milieu = (bas + haut) / 2;
        element = (void *) (((const char *) table)
                + (milieu * taille_element));
        comparaison = compare(cle, element);
        if (comparaison < 0)
            haut = milieu;
        else if (comparaison > 0)
            bas = milieu + 1;
        else
            return ((void *) element);
    }
    /* Ici, haut = bas, on n'a pas trouvé l'élément,
     * on va l'ajouter, mais nous devons vérifier de
     * quel côté de l'élément "haut".
     */
    if (haut >= (* nb_elements)) {
        element = (void *) (((const char *) table)
                + ((* nb_elements) * taille_element));
    } else {
        element = (void *) (((const char *) table)
                + (haut * taille_element));
        if (compare (cle, element) > 0) {
            element += taille_element;
            haut ++;
        }
        memmove((void *) element + taille_element,
                (void *) element,
                (* nb_elements) - haut);
```

```
    }
    memcpy((void *) element, cle, taille_element);
    (* nb_elements) ++;
    return (void *) element;
}
```

La première partie de cette fonction est calquée sur la routine `bsearch()`, implémentée dans la GlibC. Ensuite, au lieu d'échouer et de renvoyer `NULL`, elle ajoute l'élément dans la table, en le positionnant au bon endroit.

Voici un programme qui utilise cette routine pour ajouter un caractère dans une chaîne lue en argument de ligne de commande. Nous ne répétons pas l'implémentation de la routine `b_insert()`.

exemple_qsort_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    int
compare_char (const void * lm1, const void * lm2)
{
    return ((char *) lm1)[0] - ((char *) lm2)[0];
}

    int
main (int argc, char * argv[])
{
    char * table = NULL;
    int    longueur;

    if (argc != 3) {
        fprintf(stderr, "syntaxe: %s table element\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    longueur = strlen(argv[1]);
    if ((table = malloc(longueur + 2)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    strcpy(table, argv[1]);

    fprintf(stdout, "tri avec qsort ...\n");
    qsort(table, strlen(table), 1, compare_char);
    fprintf(stdout, "%s\n", table);

    fprintf(stdout, "recherche / insertion de %c\n", argv[2][0]);
    b_insert((void *) argv[2], table, & longueur, 1, compare_char);
    table[longueur] = '\0';
    fprintf(stdout, "%s\n", table);
    return EXIT_SUCCESS;
}
```

Le premier argument est une chaîne de caractères, qu'on trie avec `qsort()` après l'avoir recopiée en réservant une place supplémentaire pour l'insertion.

La routine `b_insert()` ajoute ensuite le caractère se trouvant au début du second argument, s'il ne se trouve pas déjà dans la chaîne. Voici les différents cas de figure possibles :

```
$ ./exemple_qsort_2 ertyuiop a
tri avec qsort ...
eioprtuy
recherche / insertion de a
aeioprtuy
$ ./exemple_qsort_2 ertyuiop z
tri avec qsort ...
eioprtuy
recherche / insertion de z
eioprtuyz
$ ./exemple_qsort_2 ertyuiop l
tri avec qsort ...
eioprtuy
recherche / insertion de l
eiloprtuy
$ ./exemple_qsort_2 ertyuiop i
tri avec qsort ...
eioprtuy
recherche / insertion de i
eioprtuy
$
```

Il peut parfois être gênant de réimplémenter sa propre routine pour effectuer des ajouts, et on préférerait que cette fonctionnalité soit directement incorporée dans la bibliothèque C. On peut alors se tourner vers une autre structure de données, qui est entièrement gérée par des fonctions internes de la GlibC, et qui fournit des performances remarquables en termes de complexité : les arbres binaires.

Manipulation, exploration et parcours d'un arbre binaire

Un arbre binaire est une organisation de données très répandue en algorithmique. Il s'agit d'une représentation des éléments sous forme de *nœuds*, chacun d'eux pouvant avoir 0, 1, ou 2 nœuds fils. On représente généralement les arbres binaires avec, au sommet, un nœud particulier nommé *racine*, qui n'a pas de père. Les fils d'un nœud lui sont rattachés par un lien. Un nœud sans fils est nommé *feuille*.

La *dimension* d'un arbre est égale au nombre de nœuds qui le composent, tandis que sa *profondeur* correspond à la plus grande distance qui sépare la racine d'un nœud feuille. Toute ces notions sont assez intuitives dès qu'on a assimilé que notre arbre – tel ses congénères généalogiques – pousse la tête en bas…

Les arbres binaires ordonnés présentent de surcroît la particularité suivante :

- Le fils gauche d'un nœud contient une valeur inférieure ou égale à celle de son père.
- Le fils droit d'un nœud comprend une valeur supérieure ou égale à celle de son père.

Figure 17.2
Arbre binaire

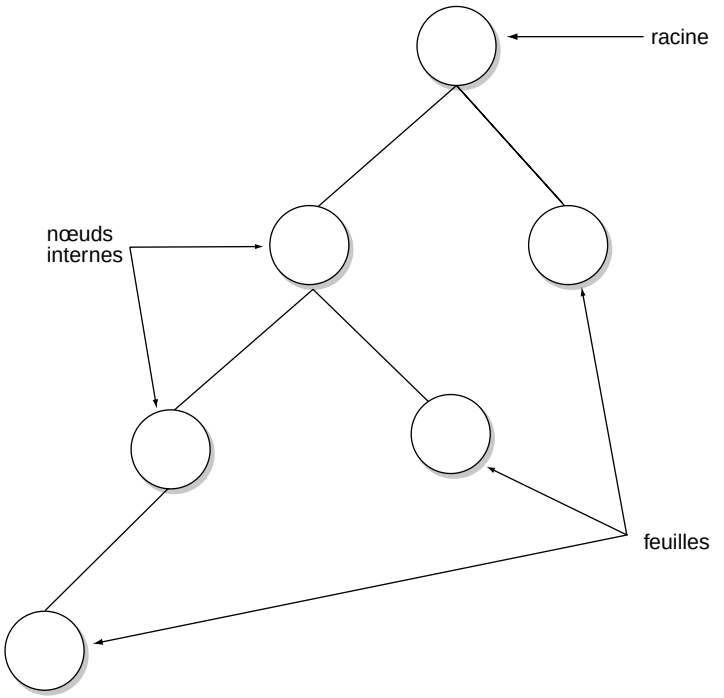
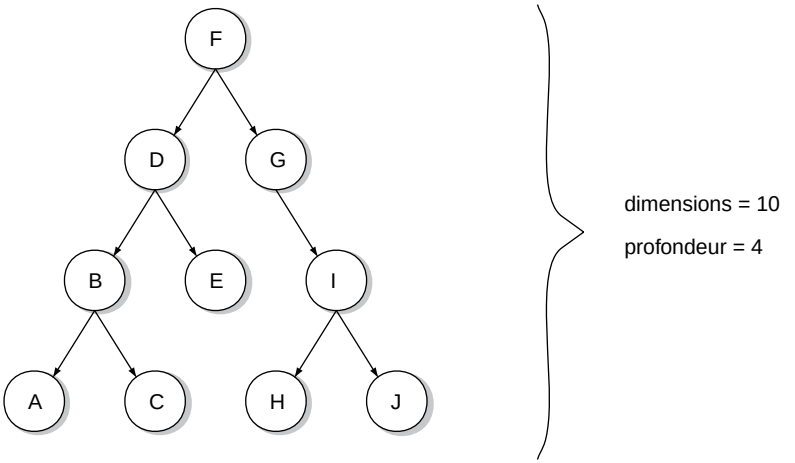


Figure 17.3
Arbre ordonné

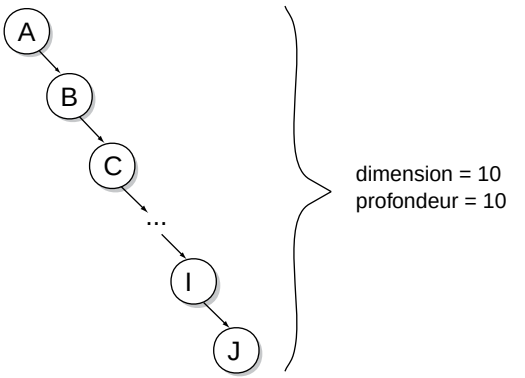


La recherche d'un élément dans un tel arbre nécessite donc, au maximum, un nombre de comparaison égal à la profondeur de l'arbre, soit *log(N)*, *N* étant sa dimension si l'arbre est correctement équilibré.

À première vue, il est aisé d'insérer une donnée, puisqu'il suffit de créer un nouveau nœud qu'on rattachera en tant que fils du dernier nœud feuille qu'on a rencontré lors de la vérification de l'existence de cet élément.

Malheureusement, cette technique n'est pas exploitable en réalité, car lors de l'ajout successif d'éléments déjà ordonnés, on va créer un arbre constitué uniquement de nœuds ayant un seul fils. L'arbre aura une profondeur égale à sa dimension, et la recherche d'un élément sera équivalente à une recherche séquentielle !

Figure 17.4
Insertion naïve d'éléments déjà ordonnés



Pour éviter cela, il est nécessaire d'équilibrer l'arbre à chaque ajout ou suppression de nœud. La bibliothèque GlibC l'effectue automatiquement en utilisant un algorithme assez compliqué fondé sur un « coloriage » des nœuds en rouge ou en noir, et vérifiant à chaque modification l'équilibre de la structure complète. L'arbre restant donc équilibré, la recherche d'un élément croît donc suivant *log(N)* lorsque *N* augmente, ce qui est presque idéal. De plus, il est possible de parcourir automatiquement tout l'arbre suivant diverses méthodes, afin de le sauvegarder de manière ordonnée par exemple.

L'arbre est représenté en interne par des structures qui ne nous concernent pas. Pour l'utiliser, nous lui transmettrons simplement des pointeurs sur nos données, convertis en pointeur `void *`. La racine de l'arbre est aussi représentée par un pointeur de type `void *`, qu'on initialise à l'origine à `NULL` avant d'insérer des éléments. Cette insertion se fait en employant la routine **`tsearch()`**, déclarée ainsi dans `<search.h>` :

```
void * tsearch (const void * cle, void ** racine,
                comparison_fn_t compare);
```

Cette routine recherche la clé transmise et, si elle ne la trouve pas, l'insère dans l'arbre. La fonction renvoie un pointeur sur l'élément trouvé ou créé, ou `NULL` si un problème d'allocation mémoire s'est présenté. On notera qu'on doit transmettre un pointeur sur la racine de l'arbre, elle-même définie comme un pointeur `void *`. En effet, la fonction peut à tout moment modifier cette racine pour réorganiser l'arbre.

Il existe une fonction **tfind()** permettant de rechercher un élément sans le créer s'il n'existe pas :

```
void * tfind (const void * cle, void ** racine, comparison_fn_t compare);
```

Si la clé n'est pas rencontrée dans l'arbre, la fonction renvoie NULL.

Pour supprimer un élément, on utilise la fonction **tdelete()**, qui assure également le rééquilibrage de l'arbre :

```
void * tdelete (const void * cle, void ** racine,
                comparison_fn_t compare);
```

L'élément est supprimé mais sa valeur est renvoyée par la fonction, sauf dans le cas où la clé n'a pas été trouvée, la routine retournant alors NULL.

Enfin, si on veut supprimer complètement un arbre, on peut utiliser la routine **tdestroy()**, qui est une extension Gnu déclarée ainsi :

```
void tdestroy (void * racine, void (* liberation) (void * element));
```

La routine liberation() sur laquelle on passe un pointeur est invoquée sur chaque nœud de l'arbre, avec en argument la valeur du nœud.

Nous avons vu les fonctions d'insertion, de recherche et de suppression d'éléments dans un arbre binaire. Nous allons à présent examiner la routine **twalk()** qui permet de parcourir l'ensemble de l'arbre en appelant une fonction de l'application sur chaque nœud. Cette fonction doit être définie ainsi :

```
void action (const void * noeud, const VISIT methode,
             const int profondeur);
```

Lorsqu'elle sera invoquée, elle recevra en premier argument un pointeur sur le nœud. Autrement dit, pour accéder aux données proprement dites, il faudra utiliser **noeud. Le second argument contient l'une des valeurs du type **enum VISIT** suivantes : leaf, preorder, postorder ou endorder, en fonction du moment où la fonction a été appelée. Nous détaillerons tout cela ci-dessous. Enfin, le troisième argument comprend la profondeur du nœud.

Le parcours se fait en profondeur d'abord, de gauche à droite. Lorsque la fonction twalk() arrive sur un nœud, elle vérifie tout d'abord s'il s'agit d'un nœud interne ou d'une feuille. Si c'est une feuille, elle appelle la routine d'action avec la méthode leaf dans le second argument, puis elle se termine.

Si c'est un nœud interne, elle invoque la routine d'action avec la méthode preorder, puis s'appelle récursivement sur le nœud fils gauche. Au retour de son fils gauche, elle appelle la routine d'action avec la méthode postorder, avant de descendre récursivement le long du fils droit. Enfin, avant de se terminer, elle invoque à nouveau la fonction d'action, avec la méthode endorder.

Pour chaque nœud interne, la routine d'action est donc appelée trois fois, et une fois pour chaque feuille de l'arbre. La fonction peut choisir d'agir ou non en fonction de la méthode avec laquelle elle a été appelée.

L'une des applications les plus pratiques est d'afficher les données lorsqu'elles se présentent sous forme leaf ou postorder. Cela permet d'obtenir la liste triée. Nous allons présenter un exemple qui construit un arbre binaire à partir de chaînes de caractères, en utilisant tsearch().

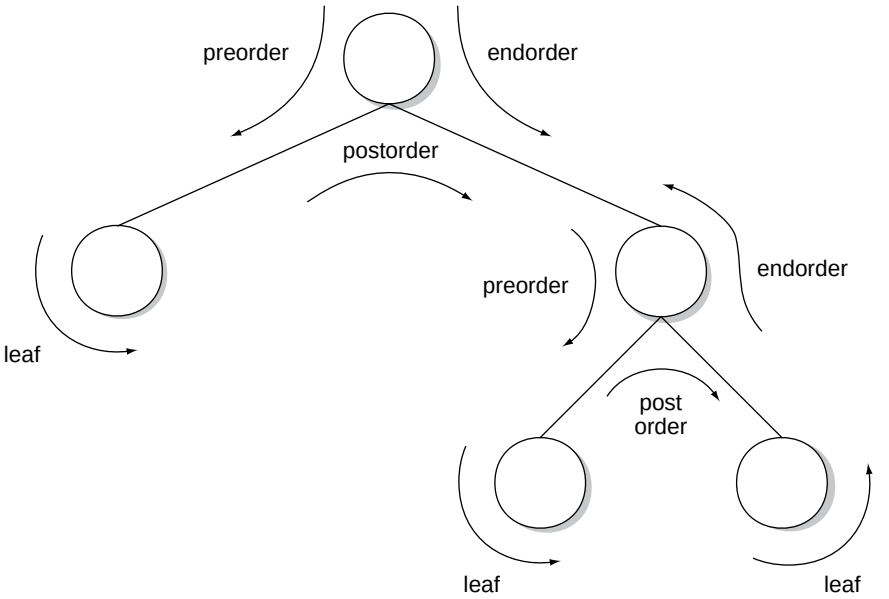


Figure 17.5
Ordre d'invocation de la routine d'action avec twalk()

Chaque chaîne ne contient qu'un seul caractère pour simplifier l'affichage. Ensuite, nous allons vérifier que les chaînes sont toutes dans le tableau en utilisant `tfind()`.

Puis, nous emploierons `twalk()` avec, à chaque fois, une sélection suivant une méthode particulière. Dans tous les cas, les feuilles `leaf` sont affichées.

exemple_tsearch.c :

```
    #include <search.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    int
compare_char (const void * lm1, const void * lm2)
{
    return strcmp(lm1, lm2);
}

    static VISIT type_parcours;

    void
parcours (const void * noeud, const VISIT methode, const int profondeur)
{
    if (methode == type_parcours)
        fprintf(stdout, "%s ", * (char **) noeud);
    else if (methode == leaf)
        fprintf(stdout, "(%s) ", * (char **) noeud);
}
```

```
    int
main (void)
{
    int     i;
    void *  racine = NULL;
    char *  chaines[] = {
    "A", "Z", "E", "R", "T", "Y", "U", "I", "O", "P",
    "Q", "S", "D", "F", "G", "H", "J", "K", "L", "M",
    "W", "X", "C", "V", "B", "N", NULL,
    };

    /* Insertion des chaines dans l'arbre binaire */
    for (i = 0; chaines[i] != NULL; i ++)
        if (tsearch(chaines[i], & racine, compare_char) == NULL) {
            perror("tsearch");
            exit(EXIT_FAILURE);
        }

    for (i = 0; chaines[i] != NULL; i ++)
        if (tfind(chaines [i], & racine, compare_char) == NULL) {
            fprintf(stderr, "%s perdue ?\n", chaines[i]);
            exit(EXIT_FAILURE);
        }

    fprintf(stdout, "Parcours preorder (+ leaf) : \n  ");
    type_parcours = preorder;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    fprintf(stdout, "Parcours postorder (+ leaf) : \n  ");
    type_parcours = postorder;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    fprintf(stdout, "Parcours endorder (+ leaf) : \n  ");
    type_parcours = endorder;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    fprintf(stdout, "Parcours leaf : \n  ");
    type_parcours = leaf;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    return EXIT_SUCCESS;
}
```

Voici le résultat de l'exécution de ce programme :

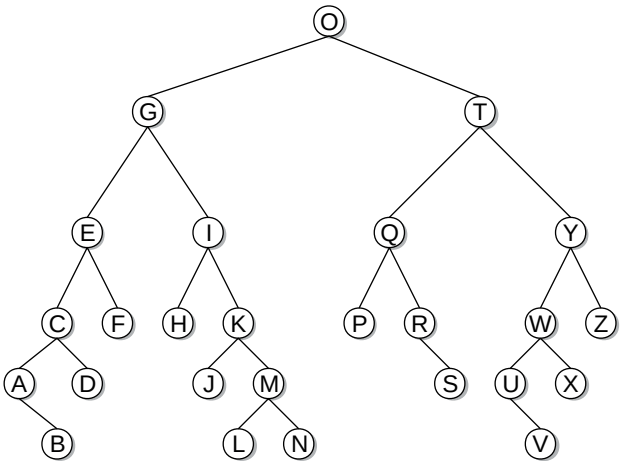
```
$ ./exemple_tsearch
Parcours preorder (+ leaf) :
  O G E C A (B) (D) (F) I (H) K (J) M (L) (N) T Q (P) R (S) Y W U (V) (X) (Z)
Parcours postorder (+ leaf) :
  A (B) C (D) E (F) G (H) I (J) K (L) M (N) O (P) Q R (S) T U (V) W (X) Y (Z)
```

```
Parcours endorder (+ leaf) :
  (B) A (D) C (F) E (H) (J) (L) (N) M K I G (P) (S) R Q (V) U (X) W (Z) Y T O
Parcours leaf :
  B D F H J L N P S V X Z
$
```

Nous vérifions bien que l'affichage ordonné des éléments s'obtient avec les méthodes `leaf` et `postorder`. On peut d'ailleurs, en observant ces résultats, retrouver la structure de l'arbre binaire interne géré par la bibliothèque C.

Figure 17.6

Arbre obtenu par insertion des lettres de l'alphabet



Nous avons donc vu ici une structure de données souple donnant de bons résultats, tant en termes d'insertion de nouveaux éléments qu'en recherche de données. Nous allons à présent étudier les routines permettant de gérer des tables de hachage, puisqu'elles peuvent en théorie offrir une complexité constante, c'est-à-dire une durée de recherche n'augmentant pas quand le nombre de données croît de manière raisonnable.

Gestion d'une table de hachage

Une table de hachage est une structure de données particulière, dans laquelle on accède directement aux éléments en calculant leur adresse à partir de leur clé. Cette organisation est particulièrement intéressante car le temps d'accès aux éléments ne dépend pas de la taille de la table. Elle est toutefois soumise à plusieurs contraintes :

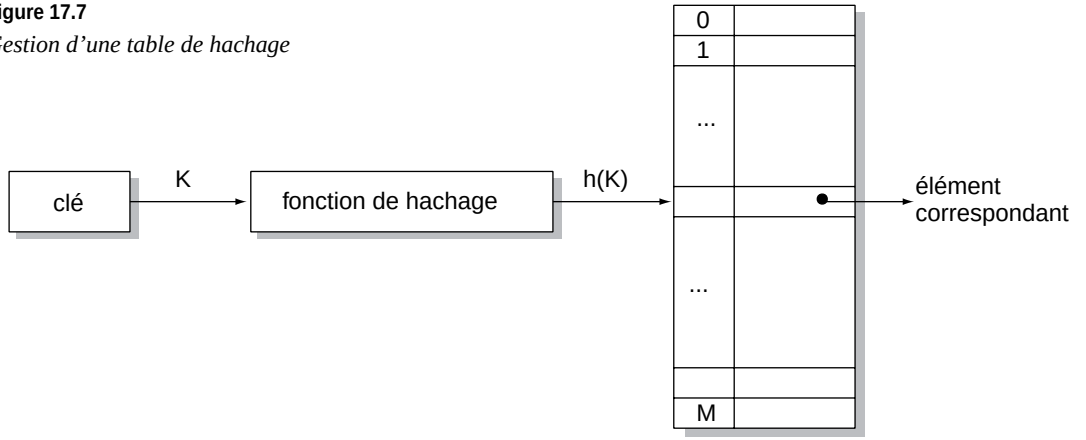
- Il faut indiquer le nombre maximal d'éléments dans la table dès sa création. Cette taille ne peut être modifiée ultérieurement.
- L'accès aux données est très efficace tant que le taux de remplissage de la table est assez faible (disons inférieur à 50 %). Les performances se dégradent par la suite et deviennent mauvaises à partir de 80 % de remplissage environ.
- Il n'est pas possible de supprimer un élément de la table. Si cette fonctionnalité est nécessaire, il faudra utiliser un indicateur dans le corps même des données, pour marquer l'élément comme « détruit ».

La table de hachage est donc une organisation idéale pour les éléments qu'on ajoute une seule fois, et dont le nombre maximal est connu dès le début. Par exemple, on utilise fréquemment une table de hachage dans les compilateurs pour stocker les mots-clés d'un langage, dans les éditeurs de liens pour mémoriser les adresses associées aux symboles, ou dans les vérificateurs orthographiques pour accéder au lexique d'une langue.

Le principe d'une table de hachage repose sur une fonction permettant de transformer la clé associée à un événement en une valeur pouvant servir d'adresse, d'indice dans une table. Cette fonction doit donc répartir les clés, le plus uniformément possible, dans l'intervalle compris entre 0 et le nombre maximal *M* d'éléments dans la table.

Les clés utilisées dans les tables de hachage gérées par la bibliothèque GlibC sont des chaînes de caractères. La bibliothèque appelle une fonction permettant de transformer la chaîne de caractères en un `unsigned int`. Elle utilise ensuite une première fonction de hachage constituée simplement de l'opération *modulo M*. La valeur résultante est employée alors comme adresse dans la table.

Figure 17.7
Gestion d'une table de hachage



Plusieurs chaînes de caractères différentes peuvent malheureusement se transformer en une valeur identique à l'issue de ce hachage. On dit alors qu'il y a une *collision* dans la table. Pour résoudre les collisions, plusieurs méthodes sont possibles :

- On utilise, pour chaque entrée dans la table, une liste de tous les éléments correspondants. Cette méthode est appelée *chaînage séparé*, elle augmente sensiblement la taille mémoire requise et le temps d'accès aux éléments. Toutefois, la taille initiale de la table n'est plus une limite stricte.
- Lors de l'insertion d'un élément, si sa place est déjà occupée, on vérifie l'emplacement suivant et, s'il est libre, on le prend. Sinon, on passe au suivant, et ainsi de suite, en revenant au début une fois la fin de la table atteinte, jusqu'à explorer toute la table. Cette méthode nommée *hachage linéaire* nécessite parfois de parcourir toute la table, et est donc inefficace lorsque le taux de remplissage est élevé.
- On peut utiliser le même principe que le hachage linéaire, mais en employant une seconde fonction de hachage pour parcourir la table, plutôt que de se déplacer d'un seul cran à

chaque fois. Cette méthode dite de *l'adressage ouvert* permet de séparer plus facilement des chaînes qui donnaient le même résultat avec la première fonction de hachage. Elle conserve toutefois les mêmes défauts lorsque le taux de remplissage augmente.

La bibliothèque GlibC est implémentée en utilisant un adressage ouvert, avec une première fonction de hachage donnant simplement le *modulo M* de la valeur numérique obtenue avec la clé, et une seconde fonction de hachage valant (1 + la valeur de hachage précédente) *modulo (M - 2)*. Cette fonction est suggérée par le paragraphe 6.4 de [KNUTH 1973c].

Pour que ces fonctions donnent des résultats satisfaisants, il est nécessaire que *M* soit un nombre premier. Lors de la création de la table, la bibliothèque C augmente donc silencieusement la valeur transmise au nombre premier le plus proche. Il ne faut donc pas s'étonner de pouvoir exceptionnellement dépasser la taille maximale de la table sans pour autant déclencher une erreur.

Nous comprenons également à présent pour quelle raison il n'est pas possible de détruire un élément d'une table de hachage, car sa présence peut avoir obligé un autre élément à se placer plus loin dans la table à la suite d'une collision. Si on libère l'emplacement, la prochaine recherche aboutira à une adresse vide, et on en conclura que la clé recherchée n'existe pas.

Pour créer une table de hachage, on utilise la fonction **hcreate()**, déclarée dans `<search.h>` :

```
int hcreate (size_t taille_maximale);
```

Cette routine crée une nouvelle table de hachage vide dans un espace de mémoire global et renvoie une valeur non nulle en cas de réussite. Si une table existe déjà ou s'il n'y a pas assez de mémoire disponible, la fonction échoue et renvoie 0.

Le fait que la table soit allouée dans une zone de mémoire globale pose deux problèmes :

- Il n'est possible d'utiliser qu'une seule table de hachage à la fois.
- L'emploi de la table par un programme multithread n'est pas sûr.

Aussi, la bibliothèque GlibC propose-t-elle, sous forme d'extension Gnu, de gérer des tables distinctes en utilisant un pointeur transmis par le programme appelant, dans lequel les éléments nécessaires au stockage sont réunis. Le type correspondant à une table de hachage est `struct hsearch_data`, et on passe aux routines un pointeur sur cette variable. Les champs internes de la structure `hsearch_data` ne sont pas documentés, aussi utilisera-t-on `memset()` pour initialiser l'ensemble de la structure à zéro.

Les routines **hcreate_r()**, **hsearch_r()** et **hdestroy_r()**, permettant de traiter des tables de hachage explicitement passées en arguments, sont donc utilisables en programmation multithread.

```
int hcreate_r (size_t taille_maximale, struct hsearch_data * table);
```

On l'utilisera donc ainsi :

```
    hsearch_data  table;

    memset (& table, 0, sizeof (table));
    if (hcreate_r (nb_elements_maxi, & table) == 0) {
        perror ("hcreate_r");
        exit (1);
    }
```

Lorsqu'on a fini d'utiliser une table, on la détruit avec **hdestroy()**

```
void hdestroy (void);
```

pour la table globale, ou

```
void hdestroy_r (struct hsearch_data * table);
```

pour les tables indépendantes.

Les éléments qu'on stocke dans une table de hachage sont du type **ENTRY**. Cette structure contient deux champs :

Type	Nom	Signification
`char *`	`key`	La clé qu'on utilise pour le hachage, consistant en une chaîne de caractères terminée par un zéro.
`char *`	`data`	Un pointeur sur une chaîne de caractères (ou sur tout autre type de données, avec la conversion `char *` pour l'initialisation). Ce pointeur est copié lorsqu'une entrée de la table est créée. Lorsqu'un élément est recherché et trouvé dans la table, un pointeur sur la structure `ENTRY` correspondante est renvoyé, contenant donc le champ `data` initial.

L'insertion ou la recherche se font avec la même fonction **hsearch()**

```
ENTRY * hsearch (ENTRY element, ACTION action);
```

ou

```
int hsearch_r (ENTRY element, ACTION action,
               ENTRY ** retour, hsearch_data * table);
```

pour la version réentrante.

ACTION est un type énuméré pouvant prendre les valeurs :

- `FIND` : pour rechercher simplement l'élément correspondant à la clé. `hsearch()` renvoie un pointeur sur l'élément de la table ayant la clé indiquée, ou `NULL` s'il n'y a pas d'élément enregistré avec cette clé. `hsearch_r()` fournit cette même information dans l'argument `retour`, qui doit être l'adresse d'un pointeur sur un enregistrement. Si la clé n'est pas trouvée, `hsearch_r()` renvoie une valeur nulle.
- `ENTER` : pour enregistrer l'élément ou mettre à jour son champ `data` s'il existe déjà. La fonction `hsearch()` renvoie un pointeur sur l'élément ajouté ou mis à jour, ou `NULL` en cas d'échec à cause d'un manque de mémoire. `hsearch_r()` fonctionne de la même manière et renvoie une valeur nulle en cas d'échec par manque de mémoire.

Nous allons mettre en pratique ces mécanismes avec un premier exemple qui va construire une table de hachage dont les enregistrements utilisent en guise de clé le libellé des jours de la semaine et des mois de l'année en français, et dont la partie `data` contient une chaîne équivalente en anglais.

exemple_hsearch.c :

```
#include <search.h>
#include <stdio.h>
#include <string.h>
```

```
    void
ajoute_entree (char * francais, char * anglais)
{
    ENTRY   entree;

    entree.key  = strdup(francais);
    entree.data = strdup(anglais);
    if (hsearch(entree, ENTER) == NULL) {
        perror("hsearch");
        exit(EXIT_FAILURE);
    }
}

    int
main (int argc, char * argv[])
{
    int     i;
    ENTRY   entree;
    ENTRY * trouve;

    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s [mois | jour]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    /* 12 mois + 7 jours */
    if (hcreate(19) == 0) {
        perror("hcreate");
        exit(EXIT_FAILURE);
    }
    ajoute_entree("janvier",       "january");
    ajoute_entree("fevrier",       "february");
    ajoute_entree("mars",          "march");
    ajoute_entree("avril",         "april");
    ajoute_entree("mai",           "may");
    ajoute_entree("juin",          "june");
    ajoute_entree("juillet",       "july");
    ajoute_entree("août",          "august");
    ajoute_entree("septembre",     "september");
    ajoute_entree("octobre",       "october");
    ajoute_entree("novembre",      "november");
    ajoute_entree("décembre",      "december");
    ajoute_entree("lundi",         "monday");
    ajoute_entree("mardi",         "tuesday");
    ajoute_entree("mercredi",      "wednesday");
    ajoute_entree("jeudi",         "thursday");
    ajoute_entree("vendredi",      "friday");
    ajoute_entree("samedi",        "satursday");
    ajoute_entree("dimanche",      "sunday");

    for (i = 1; i < argc; i ++) {
        entree.key = argv[i];
        fprintf(stdout, "%s -> ", argv[i]);
```

```
        trouve = hsearch(entree, FIND);
        if (trouve == NULL)
            fprintf(stdout, "pas dans la liste \n");
        else
            fprintf(stdout, "%s\n", trouve->data);
    }
    hdestroy();

    return EXIT_SUCCESS;
}
```

L'exécution du programme est conforme à nos attentes :

```
$ ./exemple_hsearch jeudi
jeudi -> thursday
$ ./exemple_hsearch janvier juillet dimanche samstag
janvier -> january
juillet -> july
dimanche -> sunday
samstag -> pas dans la liste
$
```

Nous allons également mettre en œuvre les extensions Gnu réentrantes, pour vérifier le fonctionnement de ces routines. L'exemple suivant fonctionne comme le précédent, avec la liste des départements français métropolitains. Nous forçons le pointeur `char * data` de la structure `ENTRY` a être manipulé comme un `int` représentant le numéro du département.

exemple_hsearch_r.c :

```
    #define _GNU_SOURCE
    #include <search.h>
    #include <stdio.h>
    #include <string.h>

    void
ajoute_entree (char * nom, int numero, struct hsearch_data * table)
{
    ENTRY   entree;
    ENTRY * retour;

    entree.key  = strdup(nom);
    entree.data = (char *) numero;
    if (hsearch_r(entree, ENTER, & retour, table) == 0) {
        perror("hsearch_r");
        exit(EXIT_FAILURE);
    }
}

    int
main (int argc, char * argv[])
{
    struct hsearch_data table;
    int     i;
    ENTRY   entree;
```

```
    ENTRY * trouve;

    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s nom-dept \n", argv[0]);
        exit(EXIT_FAILURE);
    }

    memset(& table, 0, sizeof(table));
    if (hcreate_r(100, & table) == 0) {
        perror("hcreate");
        exit(EXIT_FAILURE);
    }

    ajoute_entree("ain",                    1, & table);
    ajoute_entree("aisne",                  2, & table);
    ajoute_entree("allier",                 3, & table);
    ajoute_entree("alpes-de-haute-provence",4, & table);
    ajoute_entree("hautes-alpes",           5, & table);
    ...
    ajoute_entree("essonne",               91, & table);
    ajoute_entree("hauts-de-seine",        92, & table);
    ajoute_entree("seine-saint-denis",     93, & table);
    ajoute_entree("val-de-marne",          94, & table);
    ajoute_entree("val-d'oise",            95, & table);

    for (i = 1; i < argc; i ++) {
        entree.key = argv[i];
        fprintf(stdout, "%s -> ", argv[i]);
        if (hsearch_r(entree, FIND, & trouve, & table) == 0)
            fprintf(stdout, "pas dans la liste \n");
        else
            fprintf(stdout, "%d\n", (int) (trouve->data));
    }
    hdestroy_r(& table);

    return EXIT_SUCCESS;
}
```

Le programme s'exécute ainsi :

```
$ ./exemple_hsearch_r essonne val-de-marne seine gironde
essonne -> 91
val-de-marne -> 94
seine -> pas dans la liste
gironde -> 33
$
```

Récapitulatif sur les méthodes d'accès aux données

En définitive, nous avons vu plusieurs méthodes de structuration des informations avec l'assistance de la bibliothèque C. Nous allons essayer de dégager les avantages et les inconvénients de chacune de ces techniques. Il faut toutefois garder à l'esprit qu'une application bien

conçue ne devra pas être tributaire de telle ou telle organisation, mais pourra au contraire évoluer pour utiliser une structure mieux adaptée si le besoin s'en fait sentir. Il est aisé de regrouper dans un module distinct les appels aux routines de la bibliothèque C, en les encadrant dans des fonctions générales d'initialisation de l'ensemble des données, d'ajout, de suppression et de recherche des éléments. Il est alors possible de changer de structure pour essayer d'améliorer les performances sans avoir besoin d'intervenir sur le reste de l'application. Voici donc un récapitulatif des méthodes étudiées ici.

Recherche linéaire, table non triée

- Organisation extrêmement simple. Ajout et suppression d'éléments très faciles.
- Performances intéressantes lorsque les données sont peu nombreuses (quelques dizaines d'éléments au maximum), et lorsqu'il y a de fréquentes modifications du contenu de la table.
- La fonction de comparaison ne doit pas nécessairement fournir une relation d'ordre, mais simplement une égalité entre les éléments ; l'implémentation peut donc parfois être optimisée en ce sens.
- En contrepartie, les performances sont très mauvaises lorsque la taille de la table augmente, puisqu'il faut en moyenne balayer la moitié des éléments présents. Une amélioration est toutefois possible si certaines données, peu nombreuses, sont réclamées très fréquemment.

Recherche dichotomique, table triée

- Accès très rapide aux données, on n'effectue au maximum que $\log_2(N)$ comparaisons pour trouver un élément dans une table en contenant N.
- L'ajout d'élément est compliqué, puisqu'il faut utiliser une routine personnalisée ou retrier la table après chaque insertion.
- Les performances sont donc optimales lorsque le nombre d'éléments est important (plusieurs centaines), et si l'insertion de données est un phénomène rare. L'idéal est de pouvoir regrouper plusieurs insertions en un lot qu'on traite en une seule fois avant de retrier les données.

Arbre binaire

- Accès rapide aux données, de l'ordre de $\log_2(N)$. L'insertion ou la suppression peuvent toutefois nécessiter des réorganisations importantes de l'arbre pour conserver l'équilibre.
- Les possibilités de parcours automatique de l'arbre peuvent permettre d'implémenter automatiquement des algorithmes divers nécessitant une exploration en profondeur d'abord. On peut aussi accéder aux données triées, dans le but de les sauvegarder dans un fichier par exemple.
- Cette méthode est donc une alternative intéressante par rapport aux tables triées et à la recherche dichotomique. La recherche peut être légèrement plus longue si l'arbre n'est pas parfaitement équilibré, mais il est facile d'ajouter ou de supprimer des éléments.

Table de hachage

- Principalement utilisée pour gérer des tables de chaînes de caractères qu'on désire retrouver rapidement.

- La taille de la table est fixée dès sa création et ne peut pas être augmentée. Pour que cette méthode soit vraiment efficace, la table doit être suffisament grande pour éviter de dépasser un taux de remplissage de 60 à 80 %. L'occupation mémoire peut donc être importante.
- Il n'est pas nécessaire de fournir une fonction de comparaison, les clés sont des chaînes de caractères confrontées grâce à `strcmp()`.
- Il n'est pas possible de supprimer des éléments dans une table de hachage ni de la balayer pour en sauvegarder le contenu dans un fichier par exemple.

Conclusion

Nous verrons à nouveau des mécanismes d'organisation plus ou moins similaires dans le chapitre consacré aux bases de données conservées dans des fichiers.

Pour les lecteurs désireux d'approfondir le sujet des algorithmes de tri et voulant implémenter eux-mêmes des versions modifiées, la référence reste probablement [KNUTH 1973c] *The Art of Computer Programming* volume 3. On trouvera dans [BENTLEY 1989] *Programming Pearls*, des exemples montrant l'importance du choix d'un bon algorithme dans ce type de routines.

Les concepts fondamentaux sont présentés dans [HERNERT 2003] *Les algorithmes*. On pourra également trouver des idées intéressantes dans [MINOUX 1986] *Graphes, algorithmes, logiciels*.

Nous terminons ainsi une série de chapitres consacrés à la gestion de la mémoire d'un processus. Nous y avons étudié en détail le fonctionnement des mécanismes d'allocation et les possibilités de manipulation des chaînes de caractères et blocs de mémoire. Nous retrouverons quelques informations sur la mémoire dans le chapitre consacré aux communications entre les processus, plus particulièrement lorsque nous aborderons les segments de mémoire partagée.

18

Flux de données

Nous avons déjà abordé la notion de flux de données lors de la présentation des opérations simplifiées d'entrée-sortie pour un processus. Dans ce chapitre, nous allons étudier plus en détail la relation entre les flux de données et les descripteurs de fichiers qui leur sont associés.

Nous verrons successivement les fonctions utilisées pour ouvrir ou fermer des flux, ainsi que les routines permettant d'y écrire des données ou de s'y déplacer.

Par la suite, nous examinerons la configuration des buffers associés à un flux, ainsi que les variables indiquant leur état.

Différences entre flux et descripteurs

Une certaine confusion existe parfois dans l'esprit des programmeurs débutants sous Unix en ce qui concerne les rôles respectifs des flux de données et des descripteurs de fichiers. Il s'agit pourtant de deux notions complémentaires mais distinctes.

Les descripteurs de fichiers sont des valeurs de type `int`, que le noyau associe à un fichier à la demande d'un processus. Ces entiers sont en réalité des indices dans des tables propres à chaque processus, que le noyau est le seul à pouvoir modifier. Les descripteurs fournis par le noyau peuvent bien entendu être associés à des fichiers réguliers, mais aussi à d'autres éléments du système, comme des répertoires, des périphériques accessibles par un fichier spécial, des moyens de communication comme les tubes (*pipe*) ou les files (*FIFO*) que nous étudierons ultérieurement, ou encore des sockets utilisées pour établir la communication dans la programmation en réseau.

Les flux de données sont des objets dont le type est opaque. C'est une structure aux champs de laquelle l'application n'a pas accès. On manipule uniquement des variables pointeurs de type `FILE *`. Un flux est associé, en interne, à un descripteur de fichier, mais tout est masqué au programmeur applicatif. Un flux dispose en plus du descripteur de fichier d'une mémoire tampon, ainsi que de membres permettant de mémoriser l'état du fichier, notamment les éventuelles erreurs survenues lors des dernières opérations.

Il faut comprendre que les descripteurs de fichiers appartiennent à l'interface du noyau. Les fonctions `open()`, `close()`, `creat()`, `read()`, `write()`, `fcntl()` par exemple sont des appels-système qui dialoguent donc directement avec le noyau Linux. Les flux par contre sont une couche supérieure ajoutée aux descripteurs et qui n'appartient qu'à la bibliothèque C. Les fonctions `fopen()`, `fclose()`, `fread()` ou `fwrite()` ne sont implémentées que dans la bibliothèque C. Le noyau n'a aucune connaissance de la notion de flux.

Ceci explique d'ailleurs que la bibliothèque d'entrée-sortie du C Ansi standard ne comporte aucune indication concernant les descripteurs. Ceux-ci sont à l'origine spécifiques aux systèmes d'exploitation de type Unix et ne peuvent pas être pris en considération dans une normalisation générale. Pour assurer la portabilité d'un programme, on utilisera les flux, même si la plupart des systèmes d'exploitation courants implémentent les fonctions d'accès aux descripteurs de fichiers (pas nécessairement sous forme de primitives système d'ailleurs).

La plupart du temps, le programmeur se tournera vers les flux de données pour manipuler des fichiers, ceci pour plusieurs raisons :

- **Portabilité** : en effet, nous l'avons indiqué, les flux de données seront disponibles sur toutes les machines supportant le C standard. Ce n'est pas nécessairement vrai pour les descripteurs de fichiers.
- **Performance** : les flux utilisant des buffers pour regrouper les opérations de lecture et d'écriture, le surcoût dû à l'appel-système sur le descripteur sous-jacent est plus rare qu'avec une gestion directe du descripteur.
- **Simplicité** : la large panoplie de fonctions d'entrée et de sortie disponibles pour les flux n'existe pas pour les descripteurs de fichiers. Ceux-ci ne permettent que des lectures ou écritures de blocs mémoire complets. Il n'existe pas l'équivalent par exemple de la fonction `fgets()`, qui permet de lire une ligne de texte depuis un flux.

Les fonctions d'entrée-sortie formatées, que nous avons déjà rencontrées, comme `fprintf()` ou `fscanf()`, fonctionnent directement sur des flux. Toutefois, on peut également les employer dans un programme traitant uniquement les descripteurs de fichiers, en utilisant une chaîne de caractères intermédiaire et en appelant `sscanf()` ou `sprintf()`.

Certaines fonctionnalités sont vraiment spécifiques aux descripteurs de fichiers et ne peuvent pas être appliquées directement sur les flux. C'est le cas de la fonction `fcntl()`, qui permet de paramétrer des notions comme la lecture non bloquante, les fichiers conservés à travers un `exec()`, etc.

Il est toujours possible d'obtenir le numéro de descripteur associé à un flux, tout comme il est possible d'ouvrir un nouveau flux autour d'un descripteur donné. Le passage de l'une à l'autre des représentations des fichiers est donc possible, bien qu'à éviter pour prévenir les risques de confusion.

Nous étudierons donc en premier lieu les fonctions permettant de manipuler les flux, puisque, en général, nous les préférerons aux descripteurs.

Ouverture et fermeture d'un flux

Nous avons déjà vu un bon nombre de fonctions permettant d'échanger des données avec des flux, comme `fgetc()`, `fgets()`, `fprintf()`, etc. Lorsque nous les avons rencontrées, nous n'utilisions que les trois flux prédéfinis `stdin`, `stdout` et `stderr`, ouverts par le système avant l'exécution d'un processus.

Ouverture normale d'un flux

La fonction **fopen()**, déclarée dans <stdio.h>, permet d'ouvrir un nouveau flux à partir d'un fichier du disque :

```
FILE * fopen (const char * nom, const char * mode);
```

Cette fonction ouvre un flux à partir du fichier dont le nom est mentionné en premier argument, avec les autorisations de lecture et/ou d'écriture décrites dans une chaîne de caractères passée en second argument.

Le nom du fichier peut contenir un chemin d'accès complet ou relatif. Un chemin commençant par le caractère '/' est pris en compte à partir de la racine du système de fichiers. Sinon, le chemin d'accès commence à partir du répertoire en cours. Il faut noter que le caractère '~' représentant le répertoire personnel d'un utilisateur, est un métacaractère du shell qui n'a aucune signification pour fopen(). Si on désire situer un fichier à partir du répertoire personnel de l'utilisateur, il faut interroger la variable d'environnement HOME.

Notons également tout de suite qu'il n'est pas possible de retrouver le nom d'un fichier ouvert à partir du pointeur sur l'objet FILE (ni d'ailleurs à partir du descripteur sous-jacent). Si on désire garder une trace de ce nom, il faut le mémoriser au moment de l'ouverture.

Le mode indiqué en second argument permet de préciser le type d'accès désiré. Le mode peut prendre l'une des valeurs indiquées dans le tableau suivant :

Mode	Type d'accès
r	Lecture seule, le fichier doit exister bien entendu.
w	Écriture seule. Si le fichier existe déjà, sa taille est ramenée à zéro, sinon il est créé.
a	Écriture seule en fin de fichier. Si le fichier existe, son contenu n'est pas modifié. Sinon, il est créé.
r+	Lecture et écriture. Le contenu précédent du fichier n'est pas modifié, mais les lectures et écritures démarreront au début, écrasant les données déjà présentes.
w+	Lecture et écriture. Si le fichier existe, sa taille est ramenée à zéro, sinon il est créé.
a+	Ajout et lecture. Le contenu initial du fichier n'est pas modifié. Les lectures commenceront au début du fichier, mais les écritures se feront toujours en fin de fichier.

Sur certains systèmes non Unix, on peut rencontrer des lettres supplémentaires comme 'b' pour indiquer que le flux contient des données binaires, et non du texte. Cette précision n'est d'aucune utilité sous Linux et n'a pas d'influence sur l'ouverture du flux. Il existe également sur de nombreux systèmes des restrictions de fonctionnement pour les flux ouverts en lecture et écriture. Ces limitations n'ont pas cours sous Linux, mais nous les détaillerons, par souci de portabilité des applications, dans la section consacrée au positionnement au sein d'un flux.

Il existe une extension Gnu de fopen() qui permet d'ajouter un caractère 'x' à la fin du mode pour indiquer qu'on veut absolument créer un nouveau fichier. L'ouverture échouera si le fichier existe déjà. Cette fonctionnalité n'est pas portable, mais elle peut parfois être indispensable lorsque deux processus concurrents risquent de créer simultanément le même fichier (un verrou par exemple). Le principe consistant à tenter une ouverture en lecture seule pour vérifier si le fichier existe, suivie d'une réouverture en écriture seule s'il n'existe pas ne fonctionne pas. En effet, ces deux opérations doivent être faites de manière atomique, en un seul appel-système, sous peine de voir le noyau interrompre le processus entre les deux opérations

pour autoriser l'exécution d'un processus concurrent qui créera également le même fichier. Nous verrons qu'il y a un moyen d'ouvrir un descripteur de fichier en écriture, uniquement si le fichier n'existe pas. On pourra alors utiliser cette méthode pour ouvrir un flux autour du descripteur obtenu, afin d'implémenter l'équivalent de l'extension Gnu 'x' de `fopen()`.

La fonction `fopen()` renvoie un pointeur sur un flux, qu'on pourra ensuite utiliser dans toutes les fonctions d'entrée-sortie. En cas d'échec, `fopen()` renvoie `NULL`, et la variable globale `errno` contient le type d'erreur, qu'on peut afficher avec `perror()`. Voici quelques exemples d'ouverture de fichiers.

exemple_fopen.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    void
ouverture (char * nom, char * mode)
{
    FILE * fp;
    fprintf(stderr, "fopen(%s, %s) : ", nom, mode);
    if ((fp = fopen(nom, mode)) == NULL) {
        perror("");
    } else {
        fprintf(stderr, "Ok\n");
        fclose(fp);
    }
}

    int
main (void)
{
    ouverture("/etc/inittab", "r");
    ouverture("/etc/inittab", "w");
    ouverture("essai.fopen",  "r");
    ouverture("essai.fopen",  "w");
    ouverture("essai.fopen",  "r");
    return EXIT_SUCCESS;
}
```

La lecture du fichier `/etc/inittab` est autorisée pour tous les utilisateurs sur les distributions Linux classiques, par contre l'écriture est réservée à `root`. Le fichier `essai.fopen` n'existe pas avant l'exécution du programme. Il est créé lors de l'ouverture en mode `w`, ce qui explique que la seconde tentative d'ouverture en lecture réussisse.

```
$ ./exemple_fopen
fopen(/etc/inittab, r) : Ok
fopen(/etc/inittab, w) : Permission non accordée
fopen(essai.fopen, r) : Aucun fichier ou répertoire de ce type
fopen(essai.fopen, w) : Ok
fopen(essai.fopen, r) : Ok
$ ls essai.*
essai.fopen
$ rm essai.fopen
$
```

On notera que le nombre de fichiers simultanément ouverts par un processus est limité. Cette restriction est décrite par une constante symbolique **OPEN_MAX**. Celle-ci inclut les trois flux prédéfinis : `stdin`, `stdout` et `stderr`. Sous Linux, avec la GlibC, elle vaut 1024.

Fermeture d'un flux

La fermeture d'un flux s'effectue à l'aide de la fonction **fclose()**, dont le prototype est :

```
int fclose (FILE * flux);
```

Une fois que le flux est fermé, une tentative d'écriture ou de lecture ultérieure échouera. Le buffer alloué par la bibliothèque C lors de l'ouverture est libéré. Par contre, le buffer qu'on peut avoir explicitement installé avec la fonction `setbuf()` ou ses dérivés, que nous verrons plus bas, n'est pas libéré. La fonction `fclose()` renvoie 0 si elle réussit ou `EOF` si une erreur s'est produite.

Il est important de vérifier la valeur de retour de `fclose()`, au même titre que toutes les écritures dans un fichier. En effet, avec le principe des écritures différées, le buffer associé à un flux n'est réellement écrit dans le fichier qu'au moment de sa fermeture. Une erreur peut alors se produire si le disque est plein ou si une connexion réseau est perdue (système de fichiers NFS par exemple). Une autre erreur peut se produire avec certains types de systèmes de fichiers, comme ext3, si un problème d'entrée-sortie apparaît sur une partition, qui est alors remontée automatiquement en lecture seule. Notre flux initialement ouvert en écriture renvoie une erreur « disque plein » au moment de la fermeture.

Si la fonction `fclose()` signale une erreur, le flux n'est plus accessible, mais il est toujours possible de prévenir l'utilisateur qu'un problème a eu lieu et qu'il peut réitérer la sauvegarde après avoir arrangé la situation. On peut analyser la variable globale `errno` pour diagnostiquer le problème, les erreurs possibles étant les mêmes que pour l'appel-système `write()` que nous verrons plus loin.

On peut également fermer tous les flux ouverts par un processus avec la fonction **fcloseall()** qui est une extension Gnu déclarée dans `<stdio.h>` :

```
int fcloseall (void);
```

Normalement, tous les flux sont fermés à la fin d'un processus, mais dans certains cas – arrêt abrupt à cause d'un signal par exemple – les buffers de sortie peuvent ne pas être écrits effectivement. Il est alors possible d'appeler `fcloseall()` dans le gestionnaire de signaux concerné avant d'invoquer `abort()`.

Présentation des buffers associés aux flux

Il est temps d'étudier plus précisément les buffers associés aux flux, car ils sont souvent source de confusions.

Il existe, lors d'une écriture dans un flux, trois niveaux de buffers susceptibles de différer l'écriture. Tout d'abord, le flux est lui-même l'association d'une zone tampon et d'un descripteur de fichier. Il est possible de paramétrer le comportement de ce buffer au moyen de plusieurs fonctions que nous verrons un peu plus loin. On peut aussi forcer l'écriture du contenu d'un buffer en utilisant la fonction **fflush()**, déclarée ainsi :

```
int fflush (FILE * flux);
```

Avec cette fonction, la bibliothèque C demande au noyau d'écrire le contenu du buffer associé au flux indiqué. Elle renvoie 0 si elle réussit et `EOF` en cas d'erreur. Les erreurs sont celles qui peuvent se produire en invoquant l'appel-système `write()`.

> **Attention**
> La fonction `fflush()` n'a d'effet que sur les flux utilisés en écriture. Il est totalement illusoire de tenter d'invoquer `fflush(stdin)`, par exemple. Cet appel, malheureusement fréquemment rencontré, n'a aucune utilité et peut même déclencher une erreur sur certains systèmes (pas avec la GlibC toutefois).

Lorsqu'on appelle `fflush()`, la bibliothèque C invoque alors l'appel-système `write()` sur les données qui n'étaient pas encore transmises. Ceci se produit également lorsqu'on ferme le flux ou lorsque le buffer est plein (ou encore en fin de ligne dans certains cas). Nous reviendrons sur ces détails.

Lorsqu'un processus se termine, nous sommes donc assuré que le noyau a reçu toutes les données que nous désirions écrire dans le fichier. Les fonctions `fflush()` et `fclose()` éliminent donc tout risque d'ambiguïté si deux processus tentent d'accéder simultanément au même fichier, puisque le noyau s'interpose entre eux pour assurer la cohérence des données écrites d'un côté et lues de l'autre.

Toutefois, un deuxième niveau de buffer intervient à ce moment. Le noyau en effet implémente un mécanisme de mémoire cache pour limiter les accès aux disques. Ce mécanisme varie en fonction des systèmes de fichier utilisés (et des attributs des descripteurs de fichiers). En règle générale, le noyau diffère les écritures le plus longtemps possible. Ceci permet qu'une éventuelle modification ultérieure du même bloc de données n'ait lieu que dans la mémoire centrale, en évitant toute la surcharge due à une séquence lecture-modification-écriture sur le disque.

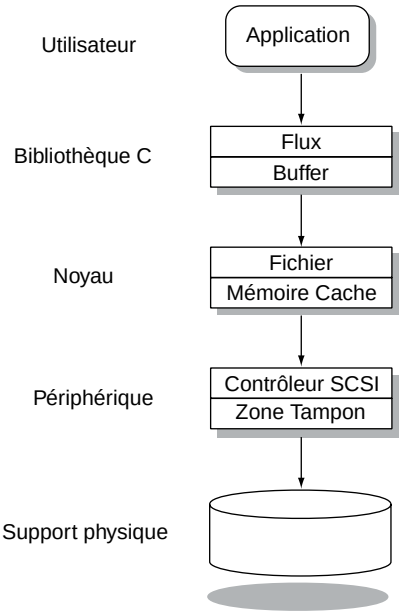
Pour s'assurer que les données sont réellement envoyées sur le disque, le noyau offre un appel-système **`sync()`**.

```
int sync (void);
```

Celui-ci transmet au contrôleur du disque les blocs de données modifiés depuis leur dernière écriture réelle. Sur d'autres systèmes Unix, l'appel-système `sync()` garantit uniquement que le noyau va commencer à mettre à niveau ses buffers, mais revient tout de suite. Depuis la version 2.0 de Linux, l'appel reste bloquant tant que tous les blocs n'ont pas été transmis sur le disque. Il existe un utilitaire `/bin/sync` qui sert uniquement à invoquer l'appel-système `sync()`. Sur les anciens Unix, on utilisait classiquement la séquence `sync; sync; sync` dans les scripts d'arrêt de la machine pour être à peu près sûr que tous les blocs en attente soient écrits sur le disque. Sous Linux, un seul appel à `/bin/sync` suffit.

Le noyau nous assure donc que lorsqu'un système de fichiers est démonté ou lorsque `sync()` revient, tous les blocs en attente auront été transmis au disque. Malheureusement, certains contrôleurs de disques (principalement SCSI) disposent de buffers internes très grands (des centaines de Mo), et rien ne garantit que les données soient immédiatement écrites physiquement. Ce point doit être pris en considération lors de la conception de systèmes informatiques basés sur des machines Linux avec des données critiques (gestion répartie, supervision de systèmes industriels ou scientifiques). On pourra alors utiliser une alimentation secourue ou des ordinateurs portables sur batterie pour garantir un certain laps de temps pour la sauvegarde physique des données en cas de défaillance secteur.

Figure 18.1
Mécanismes successifs de buffers en écriture



La fonction `fflush()` que nous avons vue ci-dessus peut également être employée avec un argument `NULL`. Dans ce cas, elle vide les buffers de tous les flux en attente d'écriture.

Ouvertures particulières de flux

La fonction `fopen()` dispose de deux variantes permettant d'ouvrir un flux de deux manières légèrement différentes. La première fonction est **`fdopen()`**, dont le prototype est :

```
FILE * fdopen (int descripteur, const char * mode);
```

Cette fonction permet de disposer d'un flux construit autour d'un descripteur de fichier déjà obtenu auparavant. Ce descripteur doit avoir été fourni précédemment par l'un des appels-système suivants :

- `open()`, `creat()`, ouvrant un fichier disque.
- `pipe()`, qui crée un tube de communication entre processus.
- `socket()`, permettant d'établir une liaison réseau.
- `dup()`, `dup2()`, qui servent à dupliquer un descripteur existant.

Nous détaillerons ces fonctions dans les chapitres à venir. Ce qu'il faut retenir pour le moment c'est la possibilité de créer un flux à partir de toutes les sources de communication offertes par le noyau Linux.

Le mode indiqué en second argument doit être compatible avec les possibilités offertes par le descripteur existant. Plus particulièrement, si le mode réclamé nécessite des accès en écriture, ceux-ci doivent être possibles sur le descripteur fourni. Il faut remarquer également que les modes `w` ou `w+` ne permettent pas dans ce cas de ramener à zéro la taille du fichier associé, car

celui-ci est déjà ouvert par le descripteur. Rappelons aussi que ce descripteur n'est pas nécessairement associé à un fichier, et que la modification de taille d'une socket réseau par exemple n'aurait pas de sens.

En cas d'échec, `fdopen()` renvoie `NULL`, sinon elle transmet un pointeur sur le flux désiré.

Cette fonction est donc principalement utile pour accéder sous forme de flux à des sources de données qu'on ne peut pas obtenir par un `fopen()` classique, comme les tubes ou les sockets. Nous en verrons plusieurs exemples dans le chapitre 28.

La seconde fonction dérivée de `fopen()` est **`freopen()`**, dont le prototype est le suivant :

```
FILE * freopen (const char * fichier, const char * mode, FILE * flux);
```

Cette fonction commence par fermer le flux indiqué en dernier argument, en ignorant toute erreur susceptible de se produire. Ensuite, elle ouvre le fichier demandé, avec le mode précisé en second argument, en utilisant le même flux que le précédent. Un pointeur sur ce dernier est renvoyé, ou `NULL` en cas d'erreur.

Étant donné que `freopen()` ne vérifie pas les erreurs susceptibles de se produire en fermant le flux original, il est indispensable d'appeler `fflush()` – et de surveiller sa valeur de retour – si le flux original a servi au préalable à écrire des données.

L'intérêt principal de cette fonction est de pouvoir rediriger les flux standard `stdin`, `stdout` et `stderr` depuis ou vers des fichiers, au sein même du programme. Tous les affichages sur `stderr` par exemple pourront ainsi être envoyés vers un fichier de débogage, sans redirection au niveau du shell. Il est aussi possible de rediriger `stderr` vers `/dev/null` pour supprimer tous les messages de diagnostic par exemple, bien que d'autres méthodes comme `syslog()` soient largement préférables. Voici un exemple de programme où on redirige la sortie standard du processus.

exemple_freopen.c :

```
    #include <stdio.h>

    int
main (void)
{
    fprintf(stdout, "Cette ligne est envoyée sur la sortie normale \n");
    if (freopen("essai.freopen", "w", stdout) == NULL) {
        perror("freopen");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Cette ligne doit se trouver dans le fichier \n");
    return EXIT_SUCCESS;
}
```

La première écriture sur `stdout` se trouve normalement affichée sur la sortie standard, la seconde est redirigée vers le fichier désiré.

```
$ ./exemple_freopen
Cette ligne est envoyée sur la sortie normale
$ cat essai.freopen
Cette ligne doit se trouver dans le fichier
$ rm essai.freopen
$
```

On notera que, les descripteurs de fichiers correspondant aux flux hérités au cours d'un appel `fork()`, la redirection est toujours valable pour le processus fils.

Avec la bibliothèque GlibC, les flux `stdin`, `stdout` et `stderr` sont des variables globales. Il serait donc tout à fait possible d'écrire :

```
fclose(stdout);
if ((stdout = fopen("essai.freopen", "w")) == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
```

Toutefois ce ne serait pas portable car de nombreuses implémentations de la bibliothèque C standard définissent `stdin`, `stdout` et `stderr` sous forme de macros. Il est donc indispensable d'utiliser `freopen()` dans ce cas.

Lectures et écritures dans un flux

Nous avons vu comment ouvrir, refermer les flux, et vider les buffers associés. Il est maintenant nécessaire d'étudier les fonctions servant à écrire effectivement ou à lire des données.

L'essentiel des fonctions d'entrée-sortie sur un flux a déjà été étudié dans le chapitre 10. Nous y avons vu successivement `fprintf()`, `vfprintf()`, `fputc()`, `fputs()` pour les écritures, `fgetc()`, `fgets()`, `fscanf()` et `vfscanf()` pour les lectures, ainsi que `fungetc()` pour rejeter un caractère dans un flux.

Nous allons ici nous intéresser aux fonctions dites d'entrée-sortie binaires. Celles-ci permettent de lire ou d'écrire le contenu intégral d'un bloc mémoire, sans se soucier de son interprétation. La fonction d'écriture est **`fwrite()`**, dont le prototype est le suivant :

```
int fwrite (const void * bloc,
            size_t taille_elements,
            size_t nb_elements,
            FILE *flux);
```

Elle permet d'écrire dans le flux indiqué un certain nombre d'éléments consécutifs, dont on indique la taille et l'adresse de départ. Pour sauvegarder le contenu d'une table d'entiers par exemple, on pourra utiliser :

```
int   table[NB_ENTIERS];
  [...]
fwrite(table, sizeof(int), NB_ENTIERS, fichier);
```

Cette fonction renvoie le nombre d'éléments correctement écrits. Si cette valeur diffère de celle qui est transmise en troisième argument lors de l'appel, une erreur s'est produite, qui doit être diagnostiquée à l'aide de la variable globale `errno`. Généralement, une telle erreur sera critique et correspondra à un problème de disque saturé ou de liaison perdue avec un système de fichiers NFS distant. Toutefois, il peut arriver que l'erreur soit bénigne, si le flux a été ouvert par la fonction `fdopen()` autour d'une socket de connexion réseau ou d'un tube de communication. Dans ces deux cas en effet, les écritures peuvent être bloquantes tant que le récepteur n'est pas disponible, et un signal peut interrompre l'appel-système `write()` sous-jacent. À cette occasion, il sera possible de réitérer l'appel de la fonction, avec les éléments non écrits correctement.

La fonction symétrique **fread()** permet de lire le contenu d'un flux et de l'inscrire dans un bloc de mémoire. Son prototype est :

```
int fread (void * bloc,
           size_t taille_elements,
           size_t nb_elements,
           FILE * flux);
```

Les arguments de `fread()` sont identiques à ceux de `fwrite()`. À ce propos, on notera que ces prototypes sont une source fréquente d'erreurs à cause de la position du pointeur `FILE *` en dernier argument, contrairement aux fonctions `fprintf()` et `fscanf()` qui le placent en premier. Le problème est que l'inversion entre le pointeur sur le flux et celui sur le bloc peut être ignorée par le compilateur si certains avertissements sont désactivés. De toute manière, il est difficile de se souvenir sans erreur des positions respectives de la taille des éléments et de leur nombre. Aussi, on s'imposera comme règle avant chaque utilisation de `fread()` ou de `fwrite()` de jeter un coup d'œil rapide sur leurs pages de manuel `fread(3)` dans une fenêtre Xterm annexe.

Comme pour `fwrite()`, la valeur de retour de `fread()` correspond au nombre d'éléments correctement lus. Par contre, à l'inverse de `fwrite()`, le nombre effectivement lu peut être inférieur à celui qui est réclamé, sans qu'une erreur critique ne se soit produite, si on atteint la fin du fichier par exemple.

Ces fonctions sont très utiles pour sauvegarder des tables de données, des structures, à condition qu'on ne les réutilise **que** sur la même machine. Les données écrites sont en effet une reproduction directe de la représentation des informations en mémoire. Cette représentation peut varier non seulement entre deux systèmes différents, par exemple en fonction de l'ordre des octets pour stocker des entiers, mais aussi sur la même machine en fonction des options utilisées par le compilateur. En voici un exemple :

exemple_enum.c :

```
    #include <stdio.h>

    typedef enum {
        un, deux, trois
    } enum_t;

    int
main (void)
{
    fprintf(stdout, "sizeof (enum_t) = %d\n", sizeof (enum_t));

    return EXIT_SUCCESS;
}
```

En fonction des options de compilation de `gcc`, la taille des données de type énuméré varie :

```
$ cc -Wall exemple_enum.c -o exemple_enum
$ ./exemple_enum
sizeof (enum_t) = 4
$ cc -Wall exemple_enum.c -o exemple_enum -fshort-enums
$ ./exemple_enum
sizeof (enum_t) = 1
$
```

Il faut donc être très prudent avec l'emploi des fonctions `fread()` et `fwrite()`, et ne les considérer que comme des moyens de sauvegarder et de récupérer des données sur une seule et même machine, sans pérennité dans le temps.

Il est conseillé dans toute application importante de prévoir des fonctionnalités d'exportation et d'importation des données moins rapides que les accès binaires directs, employant des fichiers plus volumineux mais transférables entre plusieurs systèmes hôtes ou entre diverses versions de la même application. Pour cela le plus simple est d'employer une représentation textuelle, en utilisant les fonctions `fprintf()` et `fscanf()` pour écrire et relire les données. Le meilleur exemple de cette politique est probablement le format de fichier *DXF* qui sert à exporter des dessins issus du logiciel Autocad. Ce format, documenté par l'éditeur Autodesk, représente les données sous forme de textes Ascii, donc lisibles sur l'essentiel des machines actuelles. Il est ainsi très utilisé dans les applications servant à visualiser des plans, des synoptiques, etc. Par contre, Autocad utilise en interne un format personnel *DWG*, non documenté et ne permettant pas le transfert entre machines.

Nous allons quand même présenter un exemple d'utilisation de `fread()` et de `fwrite()`, sauvegardant le contenu d'une table de structures représentant des points dans l'espace. La table est initialisée avec les points situés aux sommets d'un cube centré sur l'origine. Nous sauvegardons la table, la rechargeons, et affichons les coordonnées pour vérifier le fonctionnement.

exemple_fwrite.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    typedef struct {
        double  x;
        double  y;
        double  z;
    } point_t;

    int
main (void)
{
    point_t * table;
    int       n;
    int       i;
    FILE *    fp;

    n = 8;
    table = calloc(n, sizeof(point_t));
    if (table == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
    /* Initialisation */
    table[0].x = -1.0; table[0].y = -1.0; table[0].z = -1.0;
    table[1].x =  1.0; table[1].y = -1.0; table[1].z = -1.0;
    table[2].x = -1.0; table[2].y =  1.0; table[2].z = -1.0;
```

```
    table[3].x =  1.0; table[3].y =  1.0; table[3].z = -1.0;
    table[4].x = -1.0; table[4].y = -1.0; table[4].z =  1.0;
    table[5].x =  1.0; table[5].y = -1.0; table[5].z =  1.0;
    table[6].x = -1.0; table[6].y =  1.0; table[6].z =  1.0;
    table[7].x =  1.0; table[7].y =  1.0; table[7].z =  1.0;
    /* Sauvegarde */
    if ((fp = fopen("essai.fread", "w")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    /* Ecriture du nombre de points, suivi de la table */
    if ((fwrite(& n, sizeof(int), 1, fp) != 1)
     || (fwrite(table, sizeof(point_t), 8, fp) != 8)) {
        perror("fwrite");
        exit(EXIT_FAILURE);
    }
    fclose(fp);
    free(table);
    table = NULL;
    n = 0;
    /* Récupération */
    if ((fp = fopen("essai.fread", "r")) == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    if (fread(& n, sizeof(int), 1, fp) != 1) {
        perror("fread");
        exit(EXIT_FAILURE);
    }
    if ((table = calloc(n, sizeof(point_t))) == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
    if (fread(table, sizeof(point_t), n, fp) != 8) {
        perror("fread");
        exit(EXIT_FAILURE);
    }
    fclose(fp);
    /* Affichage */
    for (i = 0; i < n; i ++)
        fprintf(stdout, "point[%d] : % f, % f, % f \n",
                i, table[i].x, table[i].y, table[i].z);
    return EXIT_SUCCESS;
}
```

Comme on peut s'y attendre, l'exécution donne :

```
$ ./exemple_fwrite
point[0] : -1.000000, -1.000000, -1.000000
point[1] :  1.000000, -1.000000, -1.000000
point[2] : -1.000000,  1.000000, -1.000000
```

```
point[3] :  1.000000,  1.000000, -1.000000
point[4] : -1.000000, -1.000000,  1.000000
point[5] :  1.000000, -1.000000,  1.000000
point[6] : -1.000000,  1.000000,  1.000000
point[7] :  1.000000,  1.000000,  1.000000
$
```

On remarquera au passage l'emploi d'un caractère d'espacement entre le '%' et le 'f' du format de `fprintf()`, ce qui permet de conserver l'alignement des données en affichant un espace avant les valeurs positives.

Il existe deux anciennes fonctions, **`getw()`** et **`putw()`**, qu'on peut considérer comme obsolètes, permettant de lire ou d'écrire un entier. Leurs prototypes sont :

```
int getw (FILE * flux);
```

et

```
int putw (int entier, FILE * flux);
```

Ces fonctions renvoient `EOF` en cas d'échec. Sinon, elles transmettent respectivement la valeur lue et 0. Le gros défaut avec `getw()` est qu'il est impossible de distinguer une erreur de la lecture effective de la valeur `EOF` (qui vaut généralement –1). Il est donc préférable de remplacer ces deux fonctions par `fread()` et `fwrite()`.

Positionnement dans un flux

Il est rare dans une application un tant soit peu complexe qu'on ait uniquement besoin de lire les données d'un fichier séquentiellement, du début à la fin, sans jamais revenir en arrière ou sauter des portions d'informations. Il est donc naturel que la bibliothèque C mette à notre disposition des fonctions permettant de se déplacer librement dans un fichier avant de lire son contenu ou d'y écrire des données.

Ceci est géré en fait directement au niveau du descripteur de fichier, par le noyau, en mémorisant la position à laquelle se fera le prochain accès dans le fichier. Cette position est mise à jour après chaque lecture ou écriture.

Il existe trois types de fonctions pour consulter ou indiquer la position dans le fichier : le couple `ftell()` / `fseek()`, qui oblige l'indicateur de position à être de type `long int`, le couple `ftello()` / `fseeko()`, qui fonctionne de manière similaire mais sans cette restriction de type, et enfin `fgetpos()` / `fsetpos()`, qui sont encore plus portables.

Si la plupart des flux obtenus par `fopen()` depuis un fichier disque ne posent aucun problème de positionnement, ce n'est toutefois pas le cas de tous les flux possibles. Un certain nombre de sources de données sont fondamentalement séquentielles, et il n'est pas possible de se déplacer en leur sein. Tel est par exemple le cas d'un tube de communication entre processus. On ne peut y avancer qu'en lisant les données, et on ne peut en aucun cas reculer la position de lecture. Le même phénomène se produit avec les sockets de liaison réseau ou les fichiers spéciaux d'accès aux périphériques. Avec de tels flux, toute tentative de consultation ou de modification de la position courante échouera.

Positionnement classique

Les fonctions les plus simples d'accès aux positions sont certainement **ftell()** et **fseek()**, dont les prototypes sont :

```
long  ftell (FILE * flux);
```

et

```
int   fseek (FILE * flux, long position, int depart);
```

La fonction `ftell()` transmet la position courante dans le flux, mesurée en octets depuis le début du fichier. Si le positionnement n'est pas possible sur ce type de flux, `ftell()` renvoie –1.

La fonction `fseek()` permet de se déplacer dans le fichier. La position est indiquée en octets, depuis le point de départ fourni en troisième argument. Celui-ci peut prendre les valeurs suivantes :

- **SEEK_SET** (0) : on mesure la position depuis le début du fichier.
- **SEEK_CUR** (1) : le déplacement est indiqué à partir de la position courante dans le fichier.
- **SEEK_END** (2) : la position est mesurée par rapport à la fin du fichier.

Nous avons exceptionnellement indiqué entre parenthèses les valeurs des constantes symboliques définies dans `<stdio.h>`. En effet, ces constantes ne sont apparues que relativement tard, et de nombreuses applications Unix contiennent ces valeurs codées en dur dans leurs fichiers source. On peut également rencontrer les constantes obsolètes `L_SET`, `L_INCR`, `L_XTND`, qui sont des équivalentes BSD de `SEEK_SET`, `SEEK_CUR` et `SEEK_END`, définies par souci de compatibilité dans `<sys/file.h>`.

La fonction `fseek()` renvoie 0 si elle réussit, et –1 en cas d'échec.

La fonction **rewind()** permet de ramener la position courante au début du flux. Son prototype est :

```
void rewind (FILE * fp);
```

On pourrait la définir en utilisant `fseek (fichier, 0, SEEK_SET)`.

Lorsque `fseek()` ou `rewind()` sont invoquées, le contenu éventuel du buffer de sortie associé au flux est écrit dans le fichier avant le déplacement. Il existe d'ailleurs sur de nombreux systèmes une restriction à l'utilisation d'un flux en lecture et écriture. Sur de tels systèmes, une lecture ne peut suivre une opération d'écriture que si on a invoqué `fflush()`, `fseek()`, `fseeko()`, `fsetpos()` ou `rewind()` entre les deux opérations. De même, avant une écriture qui suit une lecture, il faut obligatoirement invoquer `fseek()`, `fseeko()`, `fsetpos()` ou `rewind()`. Même lorsque l'écriture doit avoir lieu exactement à la position courante résultant de la dernière lecture, il faut employer `fseek(fichier, 0, SEEK_CUR)`.

Ces limitations n'ont pas cours sous Linux. Toutefois, si un programme utilise un flux en lecture et écriture, il sera bon, par souci de portabilité, d'indiquer par un commentaire dans le fichier source les points où un `fseek()` serait obligatoire, voire de l'incorporer effectivement. Il est beaucoup plus simple de marquer ces emplacements lors de la création initiale du programme, alors qu'on maîtrise parfaitement l'utilisation du flux en question, que de rechercher, lors d'un portage, toutes les opérations ayant lieu sur le fichier, et d'analyser leur organisation pour trouver où placer les synchronisations obligatoires.

Notre premier exemple va employer `ftell()` pour repérer l'emplacement des caractères de retour à la ligne '`\n`', et afficher les longueurs successives des lignes. Nous analyserons le contenu de l'entrée standard afin de voir directement certains cas d'échec.

exemple_ftell.c :

```
#include <stdio.h>

 int
main (void)
{
    long derniere;
    long position;
    int  caractere;

    position = ftell(stdin);
    if (position == -1) {
        perror("ftell");
        exit(EXIT_FAILURE);
    }
    derniere = position;
    while ((caractere = getchar()) != EOF) {
        if (caractere == '\n') {
            position = ftell(stdin);
            if (position == -1) {
                perror("ftell");
                exit(EXIT_FAILURE);
            }
            fprintf(stdout, "%ld ", position - derniere - 1);
            derniere = position;
        }
    }

    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

Nous allons essayer de l'exécuter successivement à partir d'un fichier, depuis un tube créé par le *pipe* '`|`' du shell et depuis un fichier spécial de périphérique.

```
$ ./exemple_ftell < exemple_ftell.c
0 19 0 4 11 1 15 15 15 1 26 22 19 10 2 21 42 0 26 28 24 21 13 4 53 23 3 2 24 12 1
$ cat exemple_ftell.c | ./exemple_ftell
ftell: Repérage illégal
$ ./exemple_ftell < /dev/tty
ftell: Repérage illégal
$
```

Nous voyons que les flux obtenus à partir d'un tube ou d'un fichier spécial de périphérique ne permettent pas le positionnement.

Positionnement compatible Unix 98

Le défaut des fonctions `fseek()` et `ftell()` est de restreindre la taille d'un fichier à celle d'un `long`. Actuellement, sous Linux, un `long int` est implémenté à l'aide 4 octets. Ce qui signifie que la taille d'un fichier est limitée à $2^{31}-1$ octets (un bit est réservé pour le signe) correspondant à 2 Go. Cette dimension paraissait énorme il y a encore quelque temps, mais comme la taille des disques est régulièrement décuplée, des fichiers de 2 Go deviennent très envisageables pour stocker des séquences d'images numériques, des enregistrements sonores, ou encore des bases de données importantes.

Pour passer outre la limitation de `fseek()` et de `ftell()`, les spécifications Unix 98 ont introduit deux nouvelles fonctions, **`fseeko()`** et **`ftello()`**, utilisant un type de données spécifique, **`off_t`**. Par défaut, avec la GlibC, ce type est encore équivalent à un `long int`, mais il pourra être étendu suivant les évolutions du système.

Pour que ces fonctions soient disponibles, il faut que la constante symbolique `_XOPEN_SOURCE` soit initialisée avec la valeur `500` avant d'inclure `<stdio.h>`.

Les prototypes de ces nouvelles fonctions sont :

```
off_t  ftello (FILE * flux);
int    fseeko (FILE * flux, off_t position, int depart);
```

Leur fonctionnement est exactement le même que `ftell()` et `fseek()`, au type `off_t` près.

Nous allons créer un programme de démonstration qui servira à retourner intégralement le contenu d'un fichier. Celui-ci va uniquement utiliser des primitives `fseeko()`, `ftello()`, `fgetc()` et `fputc()`. Rappelons que nous en avons déjà construit une version bien plus efficace à l'aide de `mmap()` dans le chapitre consacré à la gestion de l'espace mémoire d'un processus.

exemple_fseeko.c :

```
    #define _XOPEN_SOURCE 500
    #include <stdio.h>
    #include <stdlib.h>

    int
main (int argc, char * argv[])
{
    int    i;
    FILE * fp;
    int    caractere;
    int    echange;
    off_t  debut;
    off_t  fin;

    if (argc < 2) {
        fprintf(stderr, "syntaxe : %s fichier...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    for (i = 1; i < argc ; i ++) {
        if ((fp = fopen(argv[i], "r+")) == NULL) {
            fprintf(stderr, "%s inaccessible \n", argv[i]);
            continue;
        }
```

```
        if (fseek(fp, 0, SEEK_END) != 0) {
            fprintf(stderr, "%s non positionnable \n", argv[i]);
            fclose(fp);
            continue;
        }
        fin = ftell(fp) - 1;
        debut = 0;
        while (fin > debut) {
            if (fseek(fp, fin, SEEK_SET) != 0)
                break;
            caractere = fgetc(fp);
            if (fseek(fp, debut, SEEK_SET) != 0)
                break;
            echange = fgetc(fp);
            if (fseek(fp, debut, SEEK_SET) != 0)
                break;
            fputc(caractere, fp);
            if (fseek(fp, fin, SEEK_SET) != 0)
                break;
            fputc(echange, fp);
            fin --;
            debut ++;
        }
        fclose(fp);
    }
    return EXIT_SUCCESS;
}
```

Nous utilisons deux pointeurs qui se rapprochent l'un de l'autre à chaque itération pour éviter d'avoir à s'interroger sur le point d'arrêt au milieu du fichier en fonction de la parité de la dimension du fichier. Nous exécutons le programme en lui demandant de retourner son propre fichier source (on reconnaît les mots-clés `return`, `fclose`, `int`, etc., à l'envers).

```
$ ./exemple_fseeko ./exemple_fseeko.c
$ cat exemple_fseeko.c

}
;SSECCUS_TIXE nruter
}
;)pf( esolcf

  [...]

tni

>h.bildts< edulcni#
>h.oidts< edulcni#
005 ECRUOS_NEPOX_ enifed#
$ ./exemple_fseeko ./exemple_fseeko.c
$ cat exemple_fseeko.c
```

```
        #define _XOPEN_SOURCE 500
        #include <stdio.h>
        #include <stdlib.h>

        int
  [...]
                fclose(fp);
        }
        return EXIT_SUCCESS;
}
$
```

Fichiers à trous

Les systèmes de fichiers utilisés par Linux en général et plus particulièrement le système ext3 gèrent les fichiers en les scindant en petits blocs dont la taille est configurable lors de la création du système de fichiers (généralement 1 Ko). Un fichier peut alors être réparti sur le disque en profitant des emplacements libres. Le noyau gère une table des blocs occupés par un fichier pour savoir où trouver les données.

Un cas particulier se présente pour les blocs uniquement remplis de zéros. Le noyau n'a pas besoin de les écrire effectivement sur le disque puisque leur contenu est constant. Aussi, de tels blocs ne sont pas réellement alloués, ils sont simplement marqués comme étant vierges dans la table associée au fichier.

Lorsqu'un processus écrit un fichier octet par octet sur le disque (comme cela peut être le cas avec l'utilitaire `cat`), le noyau ne peut pas savoir à l'avance qu'un bloc sera vierge, et il est bien obligé de lui attribuer un véritable emplacement sur le disque.

Par contre, si on décale la position d'écriture bien au-delà de la fin du fichier, le noyau sait que la zone intermédiaire est vierge par définition, et il économise des blocs en créant un trou dans le fichier. La différence entre un fichier comportant des trous et un fichier comprenant des blocs effectivement remplis par des zéros n'est pas perceptible lors de la lecture ni même lors de la consultation de la taille du fichier avec `ls`. Il faut interroger le noyau sur le volume que le fichier occupe effectivement sur le disque avec la commande `du` pour voir la différence.

Nous allons créer un petit programme qui lit son flux d'entrée standard et le copie sur la sortie standard, sans mettre les zéros, mais simplement en déplaçant l'indicateur de position dans ce cas.

exemple_fseeko_2.c :

```
    #define _XOPEN_SOURCE 500
    #include <stdio.h>
    #include <stdio.h>

    int
main (void)
{
    int   caractere;
    off_t trou;
    if (fseeko(stdout, 0, SEEK_SET) < 0) {
        fprintf(stderr, "Pas de possibilité de création de trou \n");
```

```
        while ((caractere = getchar()) != EOF)
            putchar(caractere);
        return EXIT_SUCCESS;
    }
    trou = 0;
    while ((caractere = getchar()) != EOF) {
        if (caractere == 0) {
            trou ++;
            continue;
        }
        if (trou != 0) {
            fseeko(stdout, trou, SEEK_CUR);
            trou = 0;
        }
        putchar(caractere);
    }
    if (trou != 0) {
        fseeko(stdout, trou - 1, SEEK_CUR);
        putchar(0);
    }
    return EXIT_SUCCESS;
}
```

Pour que ce programme fonctionne, il faut lui fournir en entrée un fichier contenant de larges plages de zéros (supérieures à 1 Ko). Le moyen le plus simple est d'utiliser un fichier `core`. Pour en obtenir un, nous créons un programme qui ne fait que s'envoyer à lui-même un signal `SIGSEGV`.

```
$ cat cree_core.c
    #include <signal.h>
    #include <stdlib.h>
    int
main (void)
{
    raise(SIGSEGV);
    return EXIT_SUCCESS;
}
$ ./cree_core
Segmentation fault (core dumped)
$
```

Le fichier `core` créé contient déjà des trous. Pour le vérifier nous allons le copier avec `cat` (ce qui, rappelons-le, va remplacer les trous par de véritables plages de zéros) et observer les volumes des deux fichiers.

```
$ cat < core > core.cat
$ ls -l core*
-rw-------   1 ccb      ccb     57344 Nov  9 01:02 core
-rw-rw-r--   1 ccb      ccb     57344 Nov  9 01:03 core.cat
$ du -h core*
55k     core
57k     core.cat
$
```

Finalement, nous allons utiliser notre programme de création de trous et vérifier que nous diminuons encore l'occupation du fichier.

```
$ ./exemple_fseeko_2 < core > core.trou
$ ls -l core*
-rw-------   1 ccb      ccb     57344 Nov  9 01:02 core
-rw-rw-r--   1 ccb      ccb     57344 Nov  9 01:03 core.cat
-rw-rw-r--   1 ccb      ccb     57344 Nov  9 01:05 core.trou
$ du -h core*
55k     core
57k     core.cat
42k     core.trou
$
```

Avec la taille croissante des supports de stockage actuels, l'économie de quelques blocs tient plutôt de l'anecdote que d'un réel intérêt pratique, mais il est intéressant de voir ainsi le comportement du noyau lors du déplacement en avant de la position d'écriture.

Problèmes de portabilité

Les fonctions que nous avons vues ci-dessus se comportent parfaitement bien sur un système Gnu / Linux et sur l'essentiel des systèmes Unix en général. Malgré tout, certains problèmes peuvent se poser, en particulier sur des architectures qui distinguent le stockage des données dans des fichiers binaires ou dans des fichiers de texte. Ces derniers sont parfois représentés, sur le disque, par des tables de pointeurs vers des chaînes de caractères. Le positionnement dans un tel fichier est donc repéré à la fois par le numéro de chaîne et par l'emplacement du caractère courant dans celle-ci.

Dans un tel cas, on ne peut plus considérer un fichier comme une succession linéaire de caractères ou d'octets, mais bien comme une entité dont la topologie peut s'étendre sur deux dimensions ou plus. L'utilisation du type `long int` avec `fseek()` et `ftell()`, ou du type `off_t` avec `fseeko()` et `ftello()`, n'est plus suffisante.

Pour assurer un maximum de portabilité à un programme, on se tournera vers les fonctions **`fgetpos()`** et **`fsetpos()`** :

```
int fgetpos (FILE * flux, fpos_t * position);
```

et

```
int fsetpos (FILE * flux, fpos_t * position);
```

Elles permettent de lire la position courante ou de la déplacer, en utilisant comme stockage un pointeur sur un objet de type **`fpos_t`**. Ce dernier est un type opaque, susceptible d'évoluer suivant les systèmes, les versions de bibliothèque, ou même les options de compilation.

Il n'est donc pas possible de se livrer à des calculs arithmétiques sur les déplacements mesurés par ces fonctions. La portabilité d'un programme sera assurée si on ne transmet à `fsetpos()` que des pointeurs sur des valeurs ayant été obtenues précédemment avec `fgetpos()`. Ces deux fonctions renvoient zéro si elles réussissent, une valeur non nulle sinon, et remplissent alors la variable globale `errno`.

Elles présentent malgré tout un certain nombre d'inconvénients, comme l'impossibilité de sauter directement à la fin du fichier, et nécessitent en général de mémoriser un nombre important de positions (le début du fichier, de chaque section, sous-section, enregistrement…).

C'est le prix à payer pour assurer une portabilité optimale, principalement en ce qui concerne des fichiers de texte.

Paramétrage des buffers associés à un flux

Nous avons signalé rapidement qu'un flux est une association d'un descripteur de fichier et d'un buffer de sortie, mais finalement nous n'avons pas étudié en détail ce mécanisme. Pourtant, la bibliothèque standard C offre plusieurs possibilités de paramétrage des buffers, en fonction des opérations qu'on désire effectuer sur le flux.

Type de buffers

Il existe trois types de buffers associés à un flux :

- **Buffer de bloc** : le flux dispose d'un tampon qui est rempli intégralement par les données avant qu'on invoque véritablement l'appel-système `write()` pour faire l'écriture. Un gain de temps important est alors assuré puisqu'on réduit considérablement le nombre d'appels-système à réaliser. Ce type de buffer est normalement utilisé pour tous les fichiers résidant sur le disque.
- **Buffer de ligne** : les données sont conservées dans le buffer jusqu'à ce que ce dernier soit plein, ou jusqu'à ce qu'on envoie un caractère de saut de ligne '`\n`'. Ce type de buffer est utilisé sur les flux qui sont connectés à un terminal (généralement `stderr` et `stdout`).
- **Pas de buffer** : toutes les données sont immédiatement transmises sans délai. L'appel-système `write()` est invoqué à chaque écriture.

Il est bien évident que le buffer de ligne ne présente d'intérêt que si le flux est utilisé pour transmettre du texte. Dans le cas de données binaires, le saut de ligne '`\n`' n'a pas plus de signification que tout autre caractère, et peut survenir à tout moment.

Il est toujours possible de forcer l'écriture immédiate du contenu du buffer en employant la fonction `fflush()` que nous avons vue plus haut. De même, lorsqu'on effectue une lecture sur un flux (par exemple `stdin`), tous les buffers de lignes des flux actuellement ouverts sont écrits. C'est important, par exemple pour que le message d'accueil suivant soit correctement affiché lors de la saisie, même sans retour à la ligne :

```
fprintf (stdout, "Veuillez entrer votre nom : ");
fgets (chaine, LG_CHAINE, stdin);
```

Dans ce cas, le message est écrit dans le buffer associé à `stdout`, puis, lorsque la lecture est invoquée, ce buffer est effectivement affiché, ce qui permet d'avoir un curseur placé à la suite du message pour faire la saisie.

Le flux `stdout` dispose normalement d'un buffer de ligne quand il est connecté à un terminal. Le flux `stderr` n'a pas de buffer. Les informations qu'on y écrit arrivent immédiatement sur le terminal.

Voici un petit programme d'exemple destiné à montrer que les données écrites sur `stdout` sont affichées :

- à la détection d'un saut de ligne,
- sur une demande explicite `fflush()`,
- lors d'une tentative de lecture d'un flux d'entrée,

alors que les données de `stderr` sont affichées immédiatement.

exemple_buffers.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char chaine[10];

    fprintf(stdout, "1 stdout : ligne + \\n\n");
    fprintf(stdout, "2 stdout : ligne seule");
    fprintf(stderr, "\n3 stderr : avant fflush(stdout)\n");
    fflush(stdout);
    fprintf(stderr, "\n4 stderr : après fflush (stdout)\n");
    fprintf(stdout, "5 stdout : ligne seule ");
    fprintf(stderr, "\n6 stderr : avant fgets(stdin)\n");
    fgets(chaine, 10, stdin);
    fprintf(stderr, "\n7 stderr : après fgets(stdin)\n");
    return EXIT_SUCCESS;
}
```

L'exécution donne le résultat suivant :

```
$ ./exemple_buffers
1 stdout : ligne + \n

3 stderr : avant fflush(stdout)
2 stdout : ligne seule
4 stderr : après fflush (stdout)

6 stderr : avant fgets(stdin)
5 stdout : ligne seule
                              [Entrée]

7 stderr : après fgets(stdin)
$
```

Nous voyons bien que la ligne 1 est affichée immédiatement car elle se termine par un retour à la ligne.

Mais la ligne 2 reste dans le buffer. La ligne 3 sur `stderr` apparaît tout de suite. Lorsqu'on invoque `fflush()`, la ligne 2 est effectivement affichée. La fonction `fflush()` ne revient que lorsque le buffer a réellement été vidé.

Lorsque la ligne 5 est écrite, elle reste dans le buffer. La ligne 6 est affichée immédiatement puisqu'elle arrive sur `stderr`. En demandant une lecture sur `stdin`, les buffers sont vidés et la ligne 5 est alors affichée. Nous appuyons sur la touche Entrée pour terminer la saisie.

Attention

Il ne faut pas confondre la notion de buffer de ligne, qui est interne aux flux de données, et le mode de contrôle du terminal. Lorsqu'on doit taper sur la touche « Entrée » pour valider une ligne de saisie, c'est le terminal qui gère cette ligne, et non le buffer de `stdin`. Si on veut pouvoir lire les caractères « au vol », sans attendre la touche « Entrée », il faut se pencher sur les modes de contrôle du terminal, comme nous le ferons au chapitre 33.

Modification du type et de la taille du buffer

Lorsqu'un flux utilise un buffer, la mémoire nécessaire pour celui-ci est allouée lors de la première tentative d'écriture. La taille du buffer est définie par la constante symbolique **`BUFSIZ`**, qu'on trouve dans `<stdio.h>`. Avec la GlibC, cette constante correspond à 8 Ko. Toutefois, si l'allocation échoue, la bibliothèque essaye d'obtenir un buffer de 4 Ko, puis de 2 Ko, et ainsi de suite jusqu'à la limite de 128 octets, où le mécanisme de la mémoire tampon n'a plus d'intérêt.

Lors de l'ouverture d'un flux, la bibliothèque prévoit un buffer de type bloc, sauf si le flux est connecté à un terminal, dans ce cas, le buffer est de type ligne. Le flux `stderr` représente une exception puisqu'il n'a jamais de buffer.

Nous pouvons désirer, pour de multiples raisons, modifier le type ou la contenance du buffer associé à un flux. Ceci est possible avec plusieurs fonctions, qui ont grossièrement le même effet.

La fonction la plus complète est **`setvbuf()`**, déclarée ainsi :

```
int setvbuf (FILE * flux, char * buffer, int mode, size_t taille);
```

Le premier argument est le flux sur lequel on veut agir. Le second est un pointeur sur un buffer qu'on fournit. Si ce pointeur est `NULL`, la fonction allouera elle-même une zone tampon de la taille précisée en quatrième argument. Nous préciserons les précautions à prendre lors de l'emploi d'un buffer personnalisé.

Le troisième argument correspond au type de buffer désiré. Cette valeur peut être l'une des constantes symboliques suivantes :

Nom	Signification
`_IOFBF`	(*IO Full Buffered*) indique qu'on désire un buffer de bloc.
`_IOLBF`	(*IO Line Buffered*) pour réclamer un buffer de ligne.
`_IONBF`	(*IO No Buffered*) si on ne veut aucun buffer.

Dans ce dernier cas, les second et quatrième arguments sont ignorés.

Lorsqu'on fournit un buffer personnalisé, il doit pouvoir contenir au moins la taille indiquée en dernier argument. Ce buffer sera utilisé par le flux de manière opaque, il ne faut pas tenter d'y accéder. Il est très important de vérifier que le buffer reste bien disponible tant que le flux est ouvert.

La fermeture du flux ne libère que les buffers qui ont été alloués par la bibliothèque C elle-même. Ceci inclut les buffers créés par défaut et ceux qui sont alloués lors de l'invocation de `setvbuf()` avec un second argument `NULL`.

Il y a un risque important de bogue, difficile à retrouver, lorsqu'un flux persiste à utiliser la zone mémoire qui lui a été affectée alors que celle-ci a déjà été libérée. Même la fonction `fclose()` est dangereuse si le buffer n'est plus valide. Voici un exemple de code erroné :

```
    #define TAILLE_BUFFER

    int
main (void)
{
    char   buffer[TAILLE_BUFFER];
    FILE * fp = NULL;

    fp = fopen(...)
    setvbuf(fp, buffer, _IOFBF, TAILLE_BUFFER);
    fwrite(...)
    [...]
    return EXIT_SUCCESS;
}
```

Ce code est faux car la fermeture automatique des flux ouverts se produit *après* le retour de la fonction `main()`, et donc après la libération du buffer alloué automatiquement dans la pile. Cette zone n'étant plus valide, la fonction de libération va accéder à une portion de mémoire interdite et déclencher un signal `SIGSEGV` après le retour de `main()`. La pile n'étant pas toujours gérée de la même manière suivant les systèmes d'exploitation et les compilateurs, l'erreur peut apparaître de manière totalement intempestive lors d'un portage d'application. Il faut donc être très prudent avec les buffers alloués explicitement. Il vaut mieux, autant que possible, laisser la bibliothèque C gérer l'allocation et la libération, en lui passant un pointeur `NULL`. Si ce n'est pas possible, il est préférable d'utiliser un buffer alloué dynamiquement et de s'assurer que la libération a eu lieu après la fermeture du flux.

La constante `BUFSIZE` représente une valeur qui est normalement adéquate pour tout type de buffer. Toutefois, il vaut peut-être mieux employer une valeur encore plus adaptée au fichier. Pour cela, il faut interroger le noyau en utilisant l'appel-système `stat()`. Ce dernier, qui sera détaillé dans le chapitre traitant des attributs des fichiers, remplit une structure de type `struct stat`, dont le membre **`st_blksize`** contient la taille de bloc optimale pour les entrées-sorties sur le système de fichiers utilisé. Il suffit donc de choisir une taille de buffer égale ou multiple de cette valeur :

```
    FILE *
ouvre_fichier (const char * nom)
{
    FILE *      fp = NULL;
    struct stat etat;
    int         taille_buffer = BUFSIZE;

    if ((fp = fopen(nom, "w+")) == NULL)
        return NULL;
    if (stat(nom, & etat) == 0)
        taille_buffer = etat.st_blksize;
    setvbuf(fp, NULL, _IOFBF, taille_buffer);
    return fp;
}
```

La fonction `setvbuf()` renvoie 0 si elle réussit. Sinon, le buffer précédent n'est pas modifié.

La fonction **`setbuf()`** permet uniquement de fournir un nouveau buffer, sans modifier son type, ou de supprimer toute mémoire tampon :

```
void setbuf (FILE * flux, char * buffer);
```

Si le second argument est `NULL`, le flux n'a plus de buffer. Sinon, il faut fournir un pointeur sur une zone mémoire de taille `BUFSIZ` au minimum.

Il existe deux fonctions obsolètes **`setbuffer()`** et **`setlinebuf()`**, qu'on peut parfois rencontrer, et qui sont un héritage de BSD :

```
void setbuffer (FILE * flux, char * buffer, size_t taille);
```

et

```
void setlinebuf (FILE * flux);
```

Elles peuvent toutes les deux être implémentées ainsi :

```
    void
setbuffer (FILE * flux, char * buffer, size_t taille)
{
    if (buffer == NULL)
        setvbuf(flux, NULL, _IONBF, 0);
    else
        setvbuf(flux, buffer, _IOFBF, taille);
}

    void
setlinebuf (FILE * flux)
{
    setvbuf(flux, NULL, _IOLBF, BUFSIZ);
}
```

On peut s'interroger sur la nécessité d'utiliser ces fonctions puisque la bibliothèque GlibC attribue apparemment des buffers adéquats dans toutes les situations. Voici donc quelques cas où ces fonctions se révèlent utiles :

- Une application dont le seul rôle est de filtrer des lignes de texte, à la manière de `grep` par exemple, améliore ses performances en forçant un buffer de ligne sur `stdout`, même si ce flux n'est pas connecté à un terminal. En effet, cette sortie peut être redirigée par un tube du shell vers une autre application qui finira par faire l'affichage sur le terminal. La cohérence de l'ensemble sera mieux assurée si tous les composants du tube traitent des lignes de texte en une seule fois.
- Un programme recevant des données en temps réel, sur une socket réseau par exemple, pour les traiter et les renvoyer sur sa sortie standard pourra forcer la suppression du buffer sur `stdout`, pour laisser les informations ressortir au même rythme qu'il les a reçues.
- Un processus peut employer une socket réseau pour envoyer des messages à afficher sur la console d'un administrateur. Un buffer de type ligne installé sur cette socket rendra la communication plus efficace, en évitant notamment de laisser des lignes à moitié affichées en cas de ralentissement du trafic sur le réseau.

- Enfin, nous l'avons vu, pour améliorer les performances en écriture sur un fichier disque, il est possible d'interroger le noyau pour connaître la taille de bloc optimale et de configurer un buffer binaire en conséquence.

État d'un flux

Toute opération sur un flux est susceptible de poser des problèmes, et il est important de bien vérifier les conditions de retour de chaque lecture ou écriture. La difficulté ici ne se situe pas tellement au niveau de la programmation ou de l'implémentation, mais bien plus au niveau de la conception du logiciel. L'attitude à adopter en cas de détection d'un problème sur un fichier doit être définie d'une manière homogène pour toutes les entrées-sorties de l'application. Le couple drastique `perror()` / `exit()` ne peut guère être employé que dans des petits programmes, du niveau des exemples que nous fournissons ici. L'utilisateur attend d'une application qui s'exécute dans un environnement graphique une attitude un peu plus conviviale qu'un simple arrêt abrupt à la première difficulté, d'autant que, la plupart du temps, la sortie d'erreur standard de ces applications est redirigée par le gestionnaire de fenêtres vers le fichier `.xsession-errors`, et n'est donc pas visible immédiatement. L'attitude la plus simple est souvent de proposer à l'utilisateur de recommencer la sauvegarde ou la lecture après avoir modifié le nom du fichier ou libéré de la place sur le disque.

Les conditions d'erreur en lecture sont indiquées par un retour `NULL` pour `fgets()`, par exemple, ou par un nombre d'éléments lus inférieur à celui qui est demandé avec `fread()`.

Dans ces deux cas, il n'est pas possible de distinguer immédiatement une fin de fichier normale d'une erreur plus grave (système de fichiers corrompu, liaison NFS interrompue, support amovible extrait par erreur…). Pour cela, il faut appeler l'une des fonctions **`feof()`** ou **`ferror()`** déclarées ainsi :

```
int feof (FILE * flux);
int ferror (FILE * flux);
```

La première renvoie une valeur non nulle si la fin du fichier a été atteinte, et la seconde adopte la même attitude si une autre erreur s'est produite. Dans ce cas, la variable globale `errno` peut être utilisée pour le diagnostic.

Il faut bien réaliser que ces deux fonctions n'ont de signification *qu'après* l'échec d'une lecture. Le code suivant est donc invalide :

```
    void
copie_flux_texte (FILE * flux_entree, FILE * flux_sortie)
{
    char    chaine[TAILLE_MAXI];

    while (! feof(flux_entree)) {
        fgets(chaine, TAILLE_MAXI, flux_entree);
        fputs(chaine, flux_sortie);
    }
}
```

En effet, la fin de fichier n'est détectée que lorsque la lecture a échoué. Comme elle n'a pas modifié la chaîne, qui contient la dernière ligne du fichier, celle-ci est écrite à nouveau une seconde fois. De plus, ce programme ne teste justement pas les conditions d'erreur.

En voici une version exacte, mais guère conviviale :

```
    void
copie_flux_texte (FILE * flux_entree, FILE * flux_sortie)
{
    char chaine[TAILLE_MAXI];

    while (fgets(chaine, TAILLE_MAXI, flux_entree) != NULL) {
        if (fputs(chaine, flux_sortie) == EOF) {
            perror("fputs");
            exit(EXIT_FAILURE);
        }
    }
    if (ferror(flux_entree)) {
        perror("fgets");
        exit(EXIT_FAILURE);
    }
}
```

En fait, la solution la meilleure consisterait probablement à renvoyer une valeur nulle en cas de réussite, et –1 en cas d'échec. La routine appelante pourrait alors vérifier avec `ferror()` le flux ayant posé un problème, afficher un message dans une boîte de dialogue, et proposer de recommencer après avoir modifié les noms des fichiers.

On peut effacer volontairement les indicateurs d'erreur et de fin de fichier associés à un flux. Cela se fait automatiquement lorsqu'on invoque une fonction de positionnement comme `fseek()`, `fsetpos()` ou `rewind()`, mais aussi à l'aide de la routine **`clearerr()`** :

```
void clearerr (FILE * flux);
```

Nous avons bien indiqué qu'un flux est construit, par `fopen()`, autour d'un descripteur de fichier bas niveau. Celui-ci est représenté par un `int` ayant une signification pour le noyau. Il est possible d'obtenir le numéro de descripteur associé à un flux en utilisant la fonction **`fileno()`** :

```
int fileno (FILE * flux);
```

Cette fonction renvoie le numéro du descripteur, ou –1 en cas d'échec (si le flux mentionné n'est pas valide par exemple).

Nous verrons plus tard que la fonction `fcntl()` nous permet de manipuler des paramètres importants des descripteurs de fichiers, comme la lecture non bloquante ou les verrouillages, alors que ces opérations ne sont pas possibles directement avec les flux.

Conclusion

Nous avons examiné dans ce chapitre l'essentiel des fonctionnalités concernant la manipulation des fichiers sous forme de flux.

La fonction `fileno()` nous transmet donc le numéro du descripteur de fichier associé à un flux, mais dans certains cas nous désirerons travailler directement avec ces descripteurs en employant des primitives de bas niveau, des appels-système, que nous allons étudier dans le prochain chapitre.

19

Descripteurs de fichiers

Nous analyserons dans ce chapitre les fonctions traitant directement les descripteurs de fichiers, tant du point de vue de la lecture ou écriture que pour les mécanismes plus complexes de contrôle des accès (verrouillage, lecture non bloquante…).

Nous nous retrouvons donc à un niveau plus bas que dans le chapitre précédent ; ici nous serons plus proches du noyau.

Ouverture et fermeture d'un descripteur de fichier

Un descripteur est un entier compris entre 0 et la valeur de la constante **OPEN_MAX** qui est définie dans <limits.h> (1024 sous Linux). Les descripteurs 0, 1 et 2 sont réservés respectivement pour l'entrée et la sortie standard, ainsi que pour la sortie d'erreur. Ces valeurs sont employées directement dans un si grand nombre d'applications qu'elles sont probablement immuables, mais on peut toutefois les remplacer à profit par les constantes symboliques **STDIN_FILENO**, **STDOUT_FILENO** et **STDERR_FILENO**, qui sont définies dans <unistd.h>.

Il est possible d'obtenir des descripteurs à partir d'autres éléments que des fichiers. Les appels-système pipe() ou socket() permettent d'avoir les descripteurs d'un tube de communication ou d'une liaison réseau. Nous reviendrons sur ces types de descripteurs dans les chapitres consacrés à la communication entre processus et à la programmation réseau.

Pour l'instant, nous allons nous intéresser au moyen d'obtenir un descripteur sur un fichier. Il existe pour cela deux appels-système, **open()** et **creat()**, le premier est un prototype avec un argument optionnel :

```
int open (const char * nom_fichier, int attributs, ... /* mode_t mode */);
int creat (const char * nom_fichier, mode_t mode);
```

La fonction open() prend en premier argument le nom d'un fichier à ouvrir. Le principe est le même qu'avec fopen() ; si cette chaîne commence par un '/' elle est considérée comme un

chemin débutant à la racine du système de fichiers, sinon elle est prise en compte à partir du répertoire actuel.

Le second argument est une combinaison de plusieurs éléments assemblés par un OU binaire. Tout d'abord, il faut impérativement utiliser l'une des trois constantes suivantes :

- `O_RDONLY` : fichier ouvert en lecture seule ;
- `O_WRONLY` : fichier ouvert en écriture seule ;
- `O_RDWR` : fichier ouvert à la fois en lecture et en écriture.

> **Attention**
> Il est important de bien réaliser qu'il s'agit de trois constantes indépendantes et que le mode lecture-écriture n'est **pas** une association du mode lecture seule et du mode écriture seule. La constante symbolique `O_RDWR` n'est **pas** un OU binaire entre les deux autres.

Ensuite on peut utiliser les constantes suivantes, qui permettent de préciser le mode d'ouverture :

- `O_CREAT` : pour créer le fichier s'il n'existe pas. Ceci fonctionne même avec l'ouverture `O_RDONLY`, bien que le fichier ainsi conçu reste désespérément vide. Si l'argument `O_CREAT` n'est pas mentionné, l'appel-système échoue quand le fichier n'existe pas.
- `O_EXCL` : cette constante doit être employée conjointement à `O_CREAT`. L'ouverture échouera si le fichier existe déjà. Ceci nous permet de garantir que nous venons de créer le fichier. L'appel-système étant atomique, nous sommes également assuré de ne pas entrer en conflit avec un processus concurrent tentant la même opération.
- `O_TRUNC` : si le fichier existe déjà, sa taille sera ramenée à zéro. Cette option ne doit normalement être utilisée qu'en ouverture `O_RDWR` ou `O_WRONLY`.

Enfin, les constantes suivantes sont utilisées pour paramétrer le mode de fonctionnement du fichier lors des lectures ou écritures :

- `O_APPEND` : il s'agit d'un mode d'ajout. Toutes les écritures auront lieu automatiquement en fin de fichier. Ce mode d'écriture peut aussi être modifié après l'ouverture du fichier, en utilisant l'appel-système `fcntl()`. Il ne faut pas confondre le mode d'écriture en fin de fichier et le mode d'ouverture lui-même. `O_APPEND` peut très bien être associé à `O_TRUNC` par exemple, même si cela paraît étonnant au premier abord. C'est le moyen de créer des fichiers de journalisation (comme `/var/log/messages`), qu'on réinitialise à chaque démarrage du programme. L'avantage de ce mode d'écriture est que le déplacement en fin de fichier est lié de manière atomique à l'écriture, ce qui est indispensable quand plusieurs processus doivent écrire dans le même fichier (justement dans le cas d'une journalisation). Nous reviendrons sur ce concept à la prochaine section.
- `O_NOCTTY` : si le descripteur ouvert est un terminal, il ne faut pas le prendre comme terminal de contrôle du processus, même si ce dernier n'en a pas à ce moment-là.
- `O_NONBLOCK` : cet attribut indique que les accès aux descripteurs seront non bloquants. En fait, cette option n'est jamais intéressante avec les fichiers disque, aussi son emploi avec `open()` est-il très rare. On le réserve aux files ou à certains fichiers spéciaux correspondant à des périphériques. Traditionnellement, on se sert plutôt de l'appel-système `fcntl()` pour configurer l'option de non blocage après l'ouverture des descripteurs où il peut servir

```
    }
    for (i = 1; i < argc; i ++)
        if (sscanf(argv[i], "%ld", & pid) != 1) {
            fprintf(stderr, "PID invalide : %s\n", argv[i]);
        } else {
            sid = (long) getsid((pid_t)pid);
            if (sid == -1)
                fprintf(stderr, "%ld inexistant\n", pid);
            else
                fprintf(stderr, "%ld : %ld\n", pid, sid);
        }
    return 0;
}
```

```
$ ps ax
...
  509 ?   SW   0:00 [kdm]
  521 ?   S    1:01 kwm
  538 ?   S    0:03 kbgndwm
  554 ?   S    2:36 /usr/bin/kswarm.kss -delay 3 -install -corners iiii -
  566 ?   S    0:43 kfm
  567 ?   S    0:01 krootwm
  568 ?   S    0:40 kpanel
  587 ?   SN   0:02 /usr/bin/kapm
  747 ?   SW   0:00 [axnet]
  748 ?   S   15:09 /usr/local/applix/axdata/axmain -helper
  750 ?   SW   0:00 [applix]
  758 ?   S    0:05 konsole -icon konsole.xpm -miniicon konsole.xpmi -cap
  759 ?   S    0:01 /bin/bash
  763 ?   SW   0:00 [gnome-name-serv]
$ ./exemple_getsid 0 567 748 521
0 : 759
567 : 501
748 : 501
521 : 501
$
```

Nous voyons que le processus en cours appartient à la session de son interpréteur de commandes (`/bin/bash`) et que les applications graphiques dépendent du serveur X11.

L'interaction entre un processus et un terminal s'effectue donc par l'intermédiaire de plusieurs indirections :

- Le processus appartient toujours à un groupe.
- Le groupe appartient dans son intégralité à une session.
- La session peut – éventuellement – avoir un terminal de contrôle.
- Le terminal connaît le numéro du groupe de processus en avant-plan.

C'est en général le leader de session (le shell) qui assure le basculement en avant-plan ou en arrière-plan des groupes de processus de sa session, en utilisant les fonctions de dialogue avec le terminal, `tcgetpgrp()` et `tcsetpgrp()`. Ces fonctions seront analysées ultérieurement dans le chapitre 33.

(sockets, tubes…). Les seuls cas où `O_NONBLOCK` est indispensable avec `open()` sont l'ouverture d'un fichier spécial correspondant à un port série et l'ouverture des deux extrémités d'un tube nommé dans le même processus. Nous décrirons ces deux situations dans les chapitres 30 et 33.

- **`O_SYNC`** : les écritures sur le descripteur auront lieu de manière synchronisée. Cela signifie que le noyau garantit que l'appel-système `write()` ne reviendra pas avant que les données aient été transmises au périphérique. Rappelons que, dans le cas de disques SCSI par exemple, les contrôleurs peuvent encore garder les données en mémoire tampon pendant une durée certaine avant leur écriture physique. Nous reparlerons de cette option en étudiant l'appel `write()`.

En fait, on utilise couramment `O_CREAT` et `O_TRUNC`, plus rarement `O_APPEND` et `O_EXCL`.

Pour travailler sur les ports série, on emploie souvent `O_NONBLOCK` (ou `O_NDELAY` qui est un alias obsolète), mais les autres constantes sont nettement moins sollicitées.

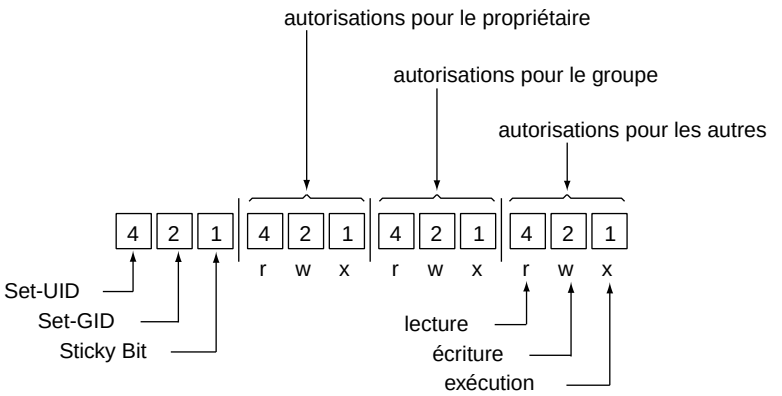
Le troisième argument de l'appel `open()` ne sert que lors d'une création de fichier. Il faut donc que l'attribut `O_CREAT` ait été indiqué. Cette valeur, de type `mode_t`, sert à signaler les autorisations d'accès au fichier nouvellement créé. On peut la fournir directement en mentionnant la valeur numérique. Celle-ci n'est lisible que dans une représentation octale, et doit donc être préfixée par un '`0`' en langage C pour être comprise comme telle par le compilateur. Il est toutefois préférable de cumuler, par l'intermédiaire d'un OU binaire, les constantes suivantes :

Constante	Valeur octale	Signification
`S_ISUID`	`04000`	Activation du bit Set-UID. Le programme s'exécutera avec l'UID effectif de son propriétaire.
`S_ISGID`	`02000`	Activation du bit Set-GID. Le programme s'exécutera avec le GID effectif de son groupe.
`S_ISVTX`	`01000`	Activation du bit « Sticky ». N'a apparemment plus d'utilité pour les fichiers réguliers de nos jours.
`S_IRUSR`	`00400`	Autorisation de lecture pour le propriétaire du fichier.
`S_IWUSR`	`00200`	Autorisation d'écriture pour le propriétaire du fichier.
`S_IXUSR`	`00100`	Autorisation d'exécution pour le propriétaire du fichier.
`S_IRWXU`	`00700`	Lecture + Écriture + Exécution pour le propriétaire du fichier.
`S_IRGRP`	`00040`	Autorisation de lecture pour le groupe du fichier.
`S_IWGRP`	`00020`	Autorisation d'écriture pour le groupe du fichier.
`S_IXGRP`	`00010`	Autorisation d'exécution pour le groupe du fichier.
`S_IRWXG`	`00070`	Lecture + Écriture + Exécution pour le groupe du fichier.
`S_IROTH`	`00004`	Autorisation de lecture pour les autres utilisateurs.
`S_IWOTH`	`00002`	Autorisation d'écriture pour les autres utilisateurs.
`S_IXOTH`	`00001`	Autorisation d'exécution pour les autres utilisateurs.
`S_IRWXO`	`00007`	Lecture + Écriture + Exécution pour les autres utilisateurs.

L'ensemble d'autorisations qu'on utilise le plus fréquemment est « `S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH` » (0644), qui permet de donner les droits de lecture à tous et les droits

d'écriture seulement au propriétaire. Un programme peut parfois créer un fichier exécutable (disons un shell script, pas obligatoirement un exécutable binaire !). Dans ce cas, il utilisera probablement les permissions « `S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH` » (0755).

Figure 19.1
Valeur octale des autorisations d'accès



Il faut être très prudent avec les autorisations qu'on accorde au groupe du fichier. En effet, les systèmes Unix peuvent adopter deux attitudes différentes lors de la création d'un fichier :

- Certains donnent au nouveau fichier le groupe effectif du processus qui le crée.
- D'autres utilisent le groupe du répertoire dans lequel le fichier est placé. Ceci permet d'assurer la cohérence de larges arborescences.

Ces deux mécanismes étant autorisés par SUSv3, il est important de vérifier, après la création d'un fichier, que son groupe est bien celui qui est attendu, si on a employé `S_IWGRP` par exemple.

Linux adopte l'attitude *a priori* la plus sage, qui consiste à utiliser le groupe effectif du processus créateur, à moins que le bit Set-GID ne soit positionné sur le répertoire d'accueil. Dans ce cas, c'est le groupe de ce dernier qui est choisi.

Le mode ainsi transmis est toutefois filtré à travers le *umask* du processus. Cette valeur, à laquelle nous verrons comment accéder dans un prochain chapitre, est retirée du mode indiqué. Ainsi, si le *umask* du processus vaut 0002, un mode 0666 sera automatiquement converti en 0664.

Il est important de fournir un argument mode lorsqu'on utilise l'option `O_CREAT` de `open()`, sinon les autorisations d'accès sont totalement imprévisibles (et généralement désastreuses).

L'appel-système `creat()`, de moins en moins utilisé, est en fait équivalent à :

```
open (nom_fichier, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

Lorsqu'on a fini d'utiliser un descripteur, on le referme à l'aide de l'appel-système **`close()`** :

```
int close (int fd);
```

Comme nous l'avions remarqué avec `fclose()`, la valeur de retour de `close()` est la dernière chance de détecter une erreur qui s'est produite durant une écriture différée dans le fichier. Si `close()` ne renvoie pas 0, le contenu du fichier est probablement inexact, et il est important de prévenir l'utilisateur, afin de recommencer la sauvegarde des données par exemple.

Un aspect déroutant des entrées-sorties de bas niveau, par rapport à la bibliothèque `<stdio.h>`, est que les prototypes et les constantes utilisés sont répartis dans une multitude de fichiers d'en-tête qui sont évidemment susceptibles de changer suivant les versions d'Unix. Pour les fonctions que nous avons étudiées, il faut inclure les fichiers suivants :

Fichier	Utilité
`<fcntl.h>`	Contient les prototypes de `open()` et de `creat()`, ainsi que les constantes `O_xxx`.
`<sys/stat.h>`	Contient les constantes de mode `S_Ixxx`.
`<sys/types.h>`	Pas obligatoire sous Linux, ce fichier peut être nécessaire sous d'autres versions d'Unix pour obtenir la définition de `mode_t`.
`<unistd.h>`	Contient la déclaration de `close()`. Cette fonction n'est en effet pas limitée aux fichiers, mais sert pour tous les descripteurs Unix.

Il est donc conseillé d'inclure systématiquement ces quatre fichiers en début de programme pour pouvoir utiliser les descripteurs avec le maximum de portabilité.

L'exemple suivant présente plusieurs tentatives d'ouverture de fichiers. Nous affichons à chaque fois les arguments employés et le résultat.

exemple_open.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/types.h>

    void
ouverture_fichier (char * nom, char * type, int attribut, mode_t mode)
{
    int fd;

    fprintf(stderr, "%s(%s) : ", nom, type);
    fd = open(nom, attribut, mode);
    if (fd < 0) {
        perror("");
    } else {
        fprintf(stderr, "Ok\n");
        close(fd);
    }
}

    int
main (void)
{
    ouverture_fichier("/etc/inittab", "O_RDONLY", O_RDONLY, 0);
    ouverture_fichier("/etc/inittab", "O_RDWR", O_RDWR, 0);
    ouverture_fichier("essai.open", "O_RDONLY", O_RDONLY, 0);
```

```
    ouverture_fichier("essai.open", "O_RDWR", O_RDWR, 0);
    ouverture_fichier("essai.open", "O_RDONLY | O_CREAT, 0640",
                            O_RDONLY | O_CREAT,
                            S_IRUSR | S_IWUSR | S_IRGRP);
    ouverture_fichier("essai.open", "O_RDWR | O_CREAT | O_EXCL, 0640",
                            O_RDWR | O_CREAT | O_EXCL,
                            S_IRUSR | S_IWUSR | S_IRGRP);
    return EXIT_SUCCESS;
}
```

Lors de l'exécution, les tentatives d'ouverture d'un fichier système dans `/etc/` ne fonctionnent évidemment qu'en lecture seule. En ce qui concerne le fichier `essai.open`, il n'est pas possible de l'ouvrir s'il n'existe pas, tant qu'on ne précise pas l'option `O_CREAT`. Par contre, dans ce cas, l'ouverture échoue si le fichier existe et qu'on a demandé l'exclusivité avec `O_EXCL`.

```
$ ./exemple_open
/etc/inittab(O_RDONLY) : Ok
/etc/inittab(O_RDWR) : Permission non accordée
essai.open(O_RDONLY) : Aucun fichier ou répertoire de ce type
essai.open(O_RDWR) : Aucun fichier ou répertoire de ce type
essai.open(O_RDONLY | O_CREAT, 0640) : Ok
essai.open(O_RDWR | O_CREAT | O_EXCL, 0640) : Le fichier existe.
$ ls -l essai.open
-rw-r-----   1 ccb      ccb       0 Nov 12 16:19 essai.open
$ rm essai.open
$
```

Nous vérifions que les droits accordés sont bien ceux qu'on a demandés. En revanche, l'exemple très simple qui suit montre l'influence de l'attribut *umask* du processus créant le fichier.

exemple_open_2.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/types.h>

    int
main (void)
{
    int fd;

    if ((fd = open("essai.open", O_RDWR | O_CREAT | O_EXCL, 0777)) < 0)
        perror("open");
    else
        close(fd);
    return EXIT_SUCCESS;
}
```

Nous demandons la création d'un fichier avec toutes les autorisations possibles.

```
$ ./exemple_open_2
$ ls -l essai.open
-rwxrwxr-x   1 ccb      ccb         0 Nov 12 16:26 essai.open
$ umask
002
$ rm essai.open
$
```

Lors de l'exécution du programme, le contenu de notre attribut *umask* est extrait des autorisations demandées, ce qui supprime le droit d'écriture pour tout le monde.

Nous avons indiqué, dans le paragraphe concernant l'ouverture d'un flux, que la bibliothèque GlibC ajoutait une extension Gnu à la fonction `fopen()`, en permettant de demander une ouverture exclusivement si le fichier n'existe pas. Ce mécanisme peut être indispensable pour s'assurer que deux processus concurrents ne risquent pas d'écrire simultanément dans le même fichier. Cette option n'étant généralement pas disponible sur d'autres environnements que la GlibC, on peut être tenté de l'implémenter naïvement ainsi :

```
    FILE *
fopen_exclusif (const char * nom_fichier, const char * mode)
{
    FILE * fp;
    if ((fp = fopen(nom_fichier, "r")) != NULL) {
        fclose(fp);
        errno = EEXIST;
        fp = NULL;
    } else {
        fp = fopen(nom_fichier, mode);
    }
    return fp;
}
```

Cette routine ne fonctionne pas car le processus peut fort bien être interrompu entre la première tentative d'ouverture, qui sert à vérifier l'existence, et l'ouverture effective du fichier. Le noyau peut alors commuter vers une autre tâche concurrente qui crée également le même fichier. Les deux processus auront l'impression d'accéder exclusivement au fichier alors que ce ne sera pas le cas.

Pour éviter ce problème, il faut s'arranger pour que la vérification d'existence et l'ouverture même soient atomiquement liées. Ceci est garanti par l'appel-système `open()` avec l'attribut `O_EXCL`. On peut alors utiliser `fdopen()` pour obtenir un flux construit autour du descripteur ainsi ouvert. Le programme suivant implémente correctement un `fopen()` exclusif.

exemple_open_3.c :

```
    #include <errno.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/types.h>
    #include <sys/stat.h>
```

```
     FILE *
fopen_exclusif (const char * nom_fichier, const char * mode_flux)
{
    int    lecture    = 0;
    int    ecriture   = 0;
    int    ajout      = 0;
    int    creation   = 0;
    int    troncature = 0;
    int    flags      = 0;
    int    i;
    int    fd;
    FILE * fp;

    for (i = 0; i < strlen(mode_flux); i ++) {
        switch (mode_flux[i]) {
            case 'a' :
                ecriture = lecture = ajout = 1;
                break;
            case 'r' :
                lecture = 1;
                break;
            case 'w' :
                ecriture = creation = troncature = 1;
                break;
            case '+' :
                ecriture = lecture = 1;
                break;
            default :
                /* soyons tolérants... on ne dit rien */
                break;
        }
    }
    if (lecture & ecriture)
        flags = O_RDWR;
    else if (lecture)
        flags = O_RDONLY;
    else if (ecriture)
        flags = O_WRONLY;
    else {
        errno = EINVAL;
        return NULL;
    }
    if (creation)
        flags |= O_CREAT;
    if (troncature)
        flags |= O_TRUNC;
    flags |= O_EXCL;
    fd = open(nom_fichier, flags, 0644);
    if (fd < 0)
        return NULL;
    fp = fdopen(fd, mode_flux);
    close(fd);
```

```
    return fp;
}

    void
ouverture (const char * nom, const char * mode, int exclusif)
{
    FILE * fp;
    fprintf(stderr, "Ouverture %s de %s, mode %s : ",
            (exclusif ? "exclusive" : ""), nom, mode);
    if (exclusif)
        fp = fopen_exclusif(nom, mode);
    else
        fp = fopen(nom, mode);
    if (fp == NULL)
        perror("");
    else {
        fprintf(stderr, "Ok\n");
        fclose(fp);
    }
}

    int
main (void)
{
    ouverture("essai.open_3", "w+", 1);
    ouverture("essai.open_3", "w+", 1);
    ouverture("essai.open_3", "w+", 0);
    return EXIT_SUCCESS;
}
```

Vérifions que les ouvertures réussissent quand le fichier n'existe pas, et qu'elles échouent sinon :

```
$ ./exemple_open_3
Ouverture exclusive de essai.open_3, mode w+ : Ok
Ouverture exclusive de essai.open_3, mode w+ : Le fichier existe.
Ouverture de essai.open_3, mode w+ : Ok
$ rm essai.open_3
$
```

Nous avons vu les principales méthodes permettant d'obtenir un descripteur de fichier. Nous examinerons ultérieurement la notion de duplication d'un descripteur, mais pour le moment nous allons nous intéresser aux primitives permettant d'en lire le contenu ou d'y écrire des données.

Lecture ou écriture sur un descripteur de fichier

Contrairement au foisonnement de fonctions qui sont mises à notre disposition par la bibliothèque C pour lire ou écrire sur un flux de données, le nombre d'appels-système manipulant les descripteurs est particulièrement concis. Il existe en tout six appels-système pour lire ou écrire des données, dont quatre sont rarement employés et sont en fait des dérivés des deux principaux. Toutefois, à cause de notre proximité avec le noyau lors de l'utilisation de ces

primitives de bas niveau, il est important de bien comprendre l'ensemble des phénomènes entrant en jeu.

Primitives de lecture

La routine de lecture la plus courante s'appelle… **read()** ! Son prototype est déclaré ainsi dans <unistd.h> :

```
ssize_t read (int descripteur, void * bloc, size_t taille);
```

Cette fonction lit dans le descripteur le nombre d'octets réclamés en troisième argument et les place dans le buffer fourni en deuxième argument. Si le descripteur permet le positionnement (par exemple un fichier disque), la lecture a lieu à l'emplacement indiqué par son indicateur de position, que nous étudierons dans une prochaine section. Ensuite, cet indicateur est augmenté du nombre d'octets lus. Si, au contraire, le descripteur ne permet pas le positionnement (port de communication série par exemple), la lecture a lieu à la position courante du descripteur. Dans ce cas, l'indicateur de positionnement n'est pas mis à jour.

L'appel-système renvoie le nombre d'octets lus. Si cette valeur correspond à la taille demandée, tout s'est bien passé. Si cette valeur est inférieure à la taille attendue mais qu'elle est positive, l'appel-système n'a pu lire qu'une partie des données voulues :

- Pour un descripteur correspondant à un fichier ordinaire, on a probablement atteint la fin du fichier.
- Pour un tube de communication, le correspondant a fermé son extrémité du tube.
- Pour une socket, le protocole réseau utilise certainement des paquets de données de taille inférieure à celle qui est réclamée.

Dans ce dernier cas, la situation est normale et se répétera probablement à chaque lecture. Par contre, dans le cas d'un tube ou d'un fichier, il est presque certain que nous sommes arrivé à la fin des données lisibles (fin du fichier ou fermeture du tube). Pour s'en assurer, la lecture suivante devrait renvoyer 0.

Si on indique une taille nulle en troisième argument, `read()` n'a aucun effet et renvoie 0.

En cas de véritable erreur, `read()` renvoie –1. Le type `ssize_t` de sa valeur de retour correspond à un `size_t` signé. La valeur maximale que peut contenir ce type de donnée est indiquée par la constante symbolique **SSIZE_MAX** définie dans <limits.h>. Avec la GlibC sur un processeur x86, elle vaut 32 767. On se limitera donc à cette dimension pour les blocs réclamés, même si la taille maximale du troisième argument de `read()` permet d'utiliser le double de valeur.

Il faut donc prendre trois cas en considération dans la valeur de retour de `read()` :

- Valeur de retour strictement positive : la lecture s'est bien passée, mais nous ne disposons que du nombre d'octets indiqué par la valeur de retour de la fonction. Si ce nombre est inférieur à la taille réclamée, ce n'est une erreur que si le contexte de l'application exige une lecture correspondant exactement à la dimension voulue.
- Valeur de retour nulle : fin de fichier ou de communication, mais pas d'erreur rencontrée jusque-là.
- Valeur de retour inférieure à zéro : une erreur s'est produite, il faut analyser la variable globale `errno`. Si cette dernière contient la valeur `EINTR`, il y a simplement eu un signal qui a interrompu l'appel-système `read()` avant qu'il ait eu le temps de lire quoi que ce soit. Dans ce cas on peut recommencer sereinement la lecture.

Dans le cas d'un descripteur correspondant à un tube, à une socket, ou à un fichier spécial de périphérique pour lequel on a demandé des lectures non bloquantes, `read()` peut également renvoyer –1, et placer `EAGAIN` dans la variable `errno` simplement si aucune donnée n'est disponible.

Dans un programme travaillant avec des liaisons réseau ou des tubes de communication – cas où `read()` est un appel-système lent –, il est ainsi fréquent d'en encadrer toutes les invocations ainsi :

```
while ((nb_octets_lus = read(fd, buffer, taille_voulue)) == -1)
    if (errno != EINTR)
        break;
```

Cette boucle permet de recommencer la lecture tant que l'appel-système est interrompu par un signal. Le problème des lectures non bloquantes est plus complexe, car on ne peut se contenter de faire une boucle `while()`, comme dans le cas de `EINTR`, au risque de voir notre programme boucler en consommant inutilement des cycles du processeur. Pour éviter cela, il existe plusieurs méthodes fondées sur les appels-système `select()` et `poll()`, que nous verrons dans le chapitre 30.

Notons que si la lecture est bloquante et si un signal interrompt `read()` alors qu'il a déjà lu quelques octets, il renverra le nombre lu, sans signaler d'erreur. Les applications faisant un large usage de signaux et de tubes de communication (ou de sockets réseau) sont souvent obligées d'implémenter un mécanisme de mémoire tampon autour de l'appel-système `read()` lorsqu'il faut lire des enregistrements constitués d'un nombre précis d'octets.

Un processus qui tente de lire depuis son terminal de contrôle alors qu'il se trouve en arrière-plan reçoit un signal `SIGTTIN`. Ce signal, par défaut, arrête le processus mais sans le tuer. Si toutefois `SIGTTIN` est ignoré, l'appel-système `read()` échoue avec l'erreur `EIO`. On peut imaginer un programme demandant un certain nombre de confirmations à l'utilisateur en fonctionnement interactif, mais désirant ignorer volontairement `SIGTTIN`, lorsqu'il est lancé en arrière-plan, pour continuer à s'exécuter comme si de rien n'était. Il devra alors utiliser des valeurs par défaut pour les saisies attendues quand `read()` déclenche l'erreur `EIO`.

La seconde fonction de lecture que nous allons étudier est **`readv()`**, qui permet de regrouper plusieurs lectures dans un seul appel. L'intérêt de cette routine est de répartir sur plusieurs zones de données le coût d'un appel-système. Cette fonction est déclarée ainsi dans `<sys/uio.h>` :

```
ssize_t readv (int descripteur,
               const struct iovec * vecteurs,
               int nombre);
```

Cette fonction lit séquentiellement les données provenant du descripteur indiqué en premier argument, et remplit les zones mémoire correspondant au nombre de vecteurs mentionné en dernier argument. Un tableau de vecteurs est transmis en second argument. Les vecteurs de type **`struct iovec`** contiennent les membres suivants :

Type	Nom	Utilisation
`void *`	`iov_base`	Un pointeur sur la zone mémoire de ce vecteur
`size_t`	`iov_len`	La longueur de cette zone mémoire

La valeur renvoyée est le nombre total d'octets lus (et pas le nombre de vecteurs remplis). Suivant les systèmes, le type de la valeur de retour de cet appel-système peut être `int` ou `ssize_t`, aussi ne faut-il pas s'étonner d'avoir un avertissement – sans conséquence – du compilateur. Les conditions d'erreur sont les mêmes que pour `read()`. On peut l'utiliser par exemple pour grouper la lecture binaire de plusieurs variables :

```
int     numero;
double  x, y, z;

struct iovec vecteur[4];

vecteur[0].iov_base = & numero;
vecteur[0].iov_len  = sizeof(int);
vecteur[1].iov_base = & x;
vecteur[1].iov_len  = sizeof(double);
vecteur[2].iov_base = & y;
vecteur[2].iov_len  = sizeof(double);
vecteur[3].iov_base = & z;
vecteur[3].iov_len  = sizeof(double);

nb_lus = readv(fd, vecteur, 4);
if (nb_lus != sizeof(int) + 3 * sizeof(double))
    return -1;
```

On pourrait effectuer le même travail avec une structure regroupant les diverses variables, mais le compilateur insère, pour aligner les champs, des octets supplémentaires susceptibles de nous compliquer la lecture. Malgré tout, cet exemple n'est certainement pas le meilleur car les lectures groupées avec `readv()` deviennent surtout performantes lorsqu'il y a un nombre important de vecteurs de petites tailles.

Il existe une autre fonction de lecture nommée **`pread()`**, assez peu utilisée. Elle a été implémentée sous Linux sous forme d'appel-système à partir du noyau 2.2. Elle est déclarée dans `<unistd.h>` :

```
ssize_t pread (int descripteur, void * bloc,
               size_t taille, off_t position);
```

Pour qu'elle soit effectivement déclarée dans `<unistd.h>`, il faut définir la constante symbolique `_XOPEN_SOURCE` et lui donner la valeur `500`, avant d'inclure le fichier d'en-tête. Cette fonction sert à implémenter les mécanismes d'entrée-sortie asynchrones que nous verrons dans le chapitre 30.

Le comportement de cette routine ainsi que sa valeur de retour sont identiques à ceux de `read()`, sauf que les données ne sont pas lues directement à la position courante dans le descripteur mais à celle qui est indiquée en quatrième argument. Cette position est mesurée en octets depuis le début du fichier. De plus, `pread()` ne modifie pas la position courante du descripteur, celle-ci restant inchangée au retour de l'appel-système. Bien entendu cette fonction échoue si le descripteur ne permet pas le positionnement (par exemple un tube). Il est important de remarquer que cette fonction, malgré son nom, n'a aucun rapport avec `popen()` et `pclose()` que nous avons analysées dans le chapitre 4.

Primitives d'écriture

Les trois appels-système d'écriture que nous allons examiner représentent le contrepoint des primitives de lecture. Nous trouvons **write()**, **writev()** et **pwrite()**, dont les prototypes sont :

```
ssize_t write  (int descripteur, const void * bloc, size_t taille);
ssize_t writev (int descripteur, const struct iovec * vecteur,
                int nombre);
ssize_t pwrite (int descripteur, const void * bloc,
                size_t taille, off_t position);
```

La fonction `write()` écrit le contenu du bloc indiqué en deuxième argument dans le descripteur fourni en premier argument. Elle renvoie le nombre d'octets effectivement écrits ou –1 en cas d'erreur. Si la taille du bloc indiquée est nulle, `write()` transmet simplement 0, sans autre effet.

Lorsque le descripteur autorise le positionnement, et s'il n'a pas l'attribut `O_APPEND`, l'écriture prend place à la position courante de celui-ci. Sinon, l'écriture a lieu à la fin du fichier.

Pour bien analyser les problèmes de déplacement au sein du fichier et de concurrence des processus, nous devons observer le mécanisme interne des entrées-sorties. Ces concepts datent des premières versions d'Unix et sont restés à peu près constants au cours des évolutions de ce système.

Un processus dispose d'une table personnelle des descripteurs ouverts. Cette table est contenue, sous Linux, dans une structure **files_struct**, définie dans le fichier d'en-tête du noyau `<linux/sched.h>`. Cette table comprend pour chaque descripteur divers attributs (comme celui de fermeture sur exécution, que nous verrons plus loin) et un pointeur sur une structure **file**, définie dans `<linux/fs.h>`.

La structure `file` comporte, entre autres, le mode d'utilisation du descripteur (lecture, écriture, ajout…) ainsi que la position courante. Elle dispose indirectement d'un pointeur sur une structure **inode** définie dans le même fichier. Entre elles s'intercale en réalité une indirection supplémentaire due à une structure **dirent**, déterminée dans `<linux/dcache.h>`, qui sert à gérer une zone de mémoire cache, qui est hors de notre propos actuel. La structure `inode` rassemble toutes les informations nécessaires à la localisation réelle du fichier sur le disque ou à l'accès aux données si le descripteur correspond à une socket ou à un tube.

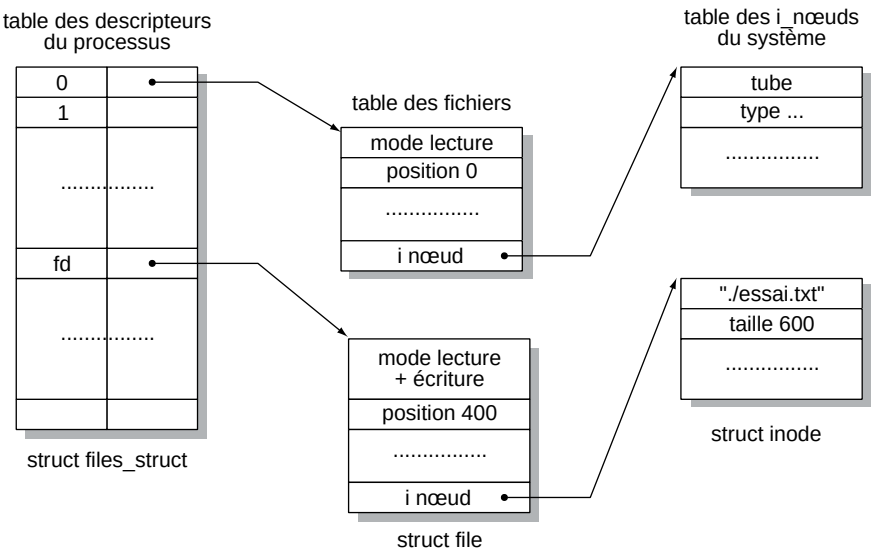
La structure `file` contient des pointeurs sur des fonctions qui, à la manière des méthodes de classe C++, implémentent les primitives d'entrée-sortie (`open`, `read`, `write`, `lseek`, `mmap`…) correspondant au type de fichier employé.

Un descripteur est donc entièrement décrit par trois niveaux de détails, qui font partie de l'implémentation traditionnelle d'Unix. Le processus comporte une table des descripteurs attribuant des numéros à chaque descripteur ouvert. Ceux-ci ont une correspondance dans la table des fichiers contenant notamment le mode d'accès et la position. Les fichiers possèdent à leur tour un correspondant dans la table des i-nœuds (*inode* ou *index node*) du système.

Nous devrons revenir sur ce mécanisme lorsque nous étudierons les possibilités de partage de fichier. Pour l'instant, nous retiendrons que la position dans un descripteur appartient à la structure `file`, alors que la longueur du fichier est contenue dans la structure `inode`.

Lorsqu'on écrit des données dans un descripteur, le noyau emploie le pointeur de fonction `write()` de la structure `file` associée pour transmettre les informations. Ensuite, il augmente la position courante de cette structure du nombre d'octets écrits. Si cette position dépasse la taille du fichier mémorisée dans l'i-nœud, celle-ci est mise à jour.

Figure 19.2
Indirections des descripteurs de fichiers



Lorsque le mode d'écriture du descripteur correspond à un ajout en fin de fichier, la position d'écriture prend d'abord la valeur de la taille de l'i-nœud associé avant de faire l'écriture. L'écriture sur un descripteur en mode `O_APPEND` est donc atomiquement constituée d'un déplacement de la position courante en fin de fichier, suivi de l'envoi effectif des données. Ceci est très important si deux processus ou plus essayent d'écrire simultanément en fin du même fichier. Ils ne risquent pas d'être interrompus entre le moment du positionnement en fin de fichier et l'écriture proprement dite, comme cela pourrait être le cas avec une implémentation naïve :

```
lseek (fd, 0, SEEK_END);
write (fd, bloc, taille);
```

L'utilisation de `O_APPEND` est donc parfaitement adaptée à la construction de fichiers de journalisation mémorisant les événements survenus dans plusieurs processus concurrents. Ceci est également vrai avec les flux ouverts en mode « `a` » ou « `a+` », qui permettent d'employer la fonction `fprintf()` pour écrire plus facilement des messages comprenant la valeur de certaines variables. Par contre, il faut savoir que deux écritures successives peuvent être séparées dans le fichier par des données provenant d'un autre processus.

Finalement, l'appel-système `write()` peut également s'assurer que les données sont immédiatement transmises au disque si le mode d'utilisation du descripteur contient l'option `O_SYNC`. Dans ce cas, le noyau demande le stockage immédiat des informations, y compris celles qui correspondent à la structure même du fichier (l'i-nœud). Bien entendu cette option diminue considérablement les performances du programme. Elle ne doit être employée que dans des cas particulièrement rares (système d'enregistrement embarqué de type « boîte noire » par exemple). Les différents processus qui accèdent à un fichier ont de toute manière une vision cohérente de celui-ci, et il n'est pas nécessaire d'utiliser `O_SYNC` pour d'autres besoins que la gestion des cas d'arrêts critiques.

Dans le même ordre d'idées, l'appel-système **fsync()** permet de demander la mise à jour des informations sur le disque de manière ponctuelle. Il est déclaré ainsi dans <unistd.h> :

```
int fsync (int fd);
```

On lui transmet simplement le descripteur du fichier à réécrire, et il renvoie 0 s'il réussit ou –1 s'il échoue. En ce cas, errno peut indiquer une erreur d'entrée-sortie grave avec le code EIO. Ceci peut arriver en cas de retrait inattendu d'un support amovible comme une disquette. L'appel-système sync(), qui ne prend pas d'argument, sert à synchroniser l'ensemble des données en attente d'écriture différée. Nous avons déjà évoqué cet appel-système en détaillant le fonctionnement des buffers associés aux flux.

Lorsque write() réussit, il transmet le nombre d'octets écrits. Si une erreur s'est produite, write() renvoie –1, et on peut analyser errno.

Il est très important de vérifier le code de retour de chaque écriture. Quel que soit le disque dur utilisé, une seule chose est à peu près sûre, c'est qu'un jour ou l'autre il sera saturé. Cela peut se produire de manière tout à fait accidentelle. Voici une anecdote qui m'est arrivée il y a quelques semaines, qui illustre bien ce cas de figure. Après une modification de configuration de l'environnement Kde, celui-ci tentait de jouer des fichiers sonores pour la plupart des événements (ouverture d'une fenêtre, mise en icône…). Malheureusement, pour une question de droit d'accès au périphérique sonore, le gestionnaire audio ne pouvait arriver à jouer ses échantillons et affichait un message sur stderr. Ce message était, comme d'habitude, redirigé vers le fichier .xsession-errors, auquel évidemment personne ne fait attention, d'autant qu'il n'apparaît pas dans un ls -l. Au bout de quelques jours de travail sans déconnexion, l'ensemble des messages d'erreur produits à chaque événement du gestionnaire de fenêtres représentait un fichier .xsession-errors de plus de 600 Mo ! La partition correspondant au répertoire /home étant déjà assez chargée, elle s'est trouvée saturée.

Le traitement de texte que j'utilisais à ce moment-là m'a indiqué que, le disque étant plein, il était obligé de désactiver les sauvegardes régulières automatiques. Cet avertissement précieux, attirant mon attention, a prouvé ainsi qu'aucune vérification du code de retour d'une écriture ne doit être négligée, y compris dans les fonctionnalités annexes comme les sauvegardes automatiques.

Une application bien conçue doit être prête à résister aux erreurs les plus farfelues de l'utilisateur :

- Tentative de sauvegarde dans un répertoire correspondant à un CD-Rom.
- Extraction inopinée d'une disquette ou d'une clé USB en cours d'écriture.
- Administrateur système débutant ayant effacé par mégarde le nœud spécial /dev/null qui sera recréé automatiquement en tant que fichier normal par la première redirection exécutée par *root*, et qui remplira peu à peu la partition racine du système de fichiers.

Ceci sans compter tous les problèmes qui peuvent se poser avec un système de fichiers monté par NFS, au gré des caprices du réseau, de l'alimentation électrique et de l'administrateur système distant.

La robustesse d'un programme dépendra donc de sa capacité à détecter au plus tôt les erreurs et à diagnostiquer correctement les problèmes pour proposer à l'utilisateur de remédier au défaut avant de recommencer la sauvegarde. La situation de la détection d'erreur au cours d'un write() est beaucoup plus cruciale que pendant un read(), car l'application est alors en

possession de données non sauvegardées, qui peuvent représenter plusieurs heures de travail et qu'il faut absolument arriver à enregistrer. Il est hors de question que tout se termine tragiquement avec un simple message d'erreur.

Les situations d'erreur susceptibles d'être repérées lors d'un appel `write()` varient en fonction du type de descripteur utilisé. Nous pouvons toutefois résumer quelques scénarios classiques à prendre en considération :

- Le système de fichiers correspondant au descripteur ouvert est saturé. L'appel `write()` renvoie le nombre d'octets qu'il a écrits. S'il n'en a écrit aucun, il renvoie –1 et `errno` contient l'erreur `ENOSPC`.
- Le fichier représenté par le descripteur a dépassé la limite maximale autorisée pour l'utilisateur. Nous verrons un exemple plus bas. L'appel `write()` renvoie le nombre d'octets écrits. Si aucun caractère n'est écrit, le processus reçoit le signal `SIGXFSZ`. Si ce signal est intercepté ou ignoré, `write()` renvoie –1 et `errno` contient `EFBIG`.
- Une erreur physique s'est produite sur le disque ou l'utilisateur a inconsidérément extrait la disquette de sauvegarde avant la fin du transfert. L'appel-système `write()` échoue donc avec une erreur `EIO`.
- Le descripteur correspond à un tube ou à une socket connectée. Le processus lecteur a fermé l'autre extrémité du tube ou la connexion réseau est rompue. Le processus reçoit alors un signal `SIGPIPE`. S'il ignore ou intercepte ce signal, `write()` renvoie –1 et `errno` contient `EPIPE`.

Tout comme nous l'avions observé avec `read()`, il existe des situations d'échec de `write()` moins tragiques que les précédentes. Dans ce cas, on peut recommencer la tentative :

- L'écriture se fait dans un descripteur de type socket ou tube, qu'on a basculé en mode non bloquant. Si le descripteur est plein, l'appel-système `write()` échoue et déclenche l'erreur `EAGAIN` en attendant qu'un processus lise les données déjà enregistrées.
- Durant l'écriture, un signal a été reçu alors qu'aucune donnée n'avait été écrite. Dans ce cas, `write()` renvoie –1 et place `EINTR` dans `errno`.
- On tente d'écrire dans une portion de fichier sur laquelle un autre processus vient de placer un verrouillage strict, comme nous le verrons plus loin. L'écriture échoue alors avec une erreur `EAGAIN`.

On remarquera qu'en cas d'échec de `write()` avec une erreur `EAGAIN`, il est probablement inutile de réessayer immédiatement l'écriture. Il vaut mieux laisser un peu de temps au processus lecteur pour vider le tube plein, ou à celui qui a verrouillé le fichier pour écrire ses données. On évitera donc de faire une boucle du type :

```
while ((nb_ecrits = write(fd, buffer, taille)) == -1)
    if ((errno != EINTR) && (errno != EAGAIN))
        break;
```

Cette boucle consomme inutilement du temps processeur. Il est préférable que le processus appelant se mette quelques instants en sommeil, en cas d'erreur `EAGAIN`, avant de recommencer sa tentative. Le code suivant est déjà préférable.

```
while ((nb_ecrits = write(fd, buffer, taille)) == -1) {
    if (errno == EINTR)
        continue;
```

```
    if (errno != EAGAIN)
        break;
    sleep(1);
}
```

Une autre solution encore plus performante peut être construite autour de l'appel-système `select()`, que nous étudierons dans le chapitre 30.

Théoriquement, `write()` ne peut pas renvoyer une valeur nulle, sauf si on lui a demandé explicitement d'écrire 0 octet. Si l'appel-système a pu écrire quelques caractères avant qu'une erreur se produise, il renvoie ce nombre d'octets, sinon il renvoie –1. Si un programme est susceptible de recevoir des signaux tout en employant des appels-système `write()` pouvant bloquer (sockets, tubes…), il faut construire une boucle permettant d'envoyer toutes les données, éventuellement en plusieurs fois.

On peut utiliser par exemple un code du genre :

```
    ssize_t
mon_write (int fd, const void * buffer, size_t taille)
{
    const void *    debut   = buffer;
    size_t          restant = taille;
    ssize_t         ecrits  = 0;

    while (restant > 0) {
        while ((ecrits = write(fd, debut, restant)) == -1) {
            if (errno == EINTR)
                continue;
            if (errno !=EAGAIN)
                return -1;
            sleep(1);
        }
        restant -= ecrits;
        debut += ecrits;
    }
    return taille;
}
```

Ceci, rappelons-le, ne concerne que des écritures se faisant dans des descripteurs susceptibles de bloquer (sockets, tubes, files…) alors que le processus risque de recevoir des signaux utilisés par l'application.

L'exemple suivant va mettre en relief le comportement de `write()` lors d'une tentative de dépassement de la taille maximale autorisée pour un fichier. Nous allons d'abord réduire la limite à une valeur plus faible et tenter des écritures successives. Nous restreignons la limite `FSIZE` à une valeur qui n'est pas un multiple de la taille du buffer écrit, afin d'obtenir en premier lieu un nombre d'octets écrits inférieur à celui qui est attendu. À la tentative suivante, `write()` échouera en déclenchant d'ailleurs le signal `SIGXFSZ`.

exemple_write.c :

```
#define _GNU_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
```

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <sys/resource.h>
    #include <sys/stat.h>
    #include <sys/types.h>

    #define TAILLE_BLOC 1024
    #define DEPASSEMENT  767

    void
gestionnaire (int numero)
{
    fprintf(stderr, "Signal %d reçu : %s\n", numero, strsignal(numero));
}

    int
main (void)
{
    struct rlimit limite;
    int           fd;
    char          bloc[TAILLE_BLOC];
    int           nb_ecrits;

    signal(SIGXFSZ, gestionnaire);

    if (getrlimit(RLIMIT_FSIZE, & limite) != 0) {
        perror("getrlimit");
        exit(EXIT_FAILURE);
    }
    limite.rlim_cur = 3 * TAILLE_BLOC + DEPASSEMENT;
    if (setrlimit(RLIMIT_FSIZE, & limite) != 0) {
        perror("setrlimit");
        exit(EXIT_FAILURE);
    }
    fd = open("essai.write", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    memset(bloc, 1, TAILLE_BLOC);
    do {
        nb_ecrits = write(fd, bloc, TAILLE_BLOC);
        if (nb_ecrits != TAILLE_BLOC) {
            fprintf(stderr, "nb_ecrits = %d\n", nb_ecrits);
            if (errno != 0) {
                fprintf(stderr, "errno = %d : ", errno);
                perror("");
            }
        }
```

```
    } while (nb_ecrits != -1);
    close(fd);
    return EXIT_SUCCESS;
}
```

L'exécution suivante montre bien les deux appels `write()` qui échouent : le premier n'écrit que 767 octets au lieu des 1 024 attendus. Le second appel déclenche `SIGXFSZ` et renvoie –1, et `errno` est correctement remplie.

```
$ ./exemple_write
nb_ecrits = 767
Signal 25 reçu : Débordement de la taille permise pour un fichier
nb_ecrits = -1
errno = 27 : Fichier trop gros
$
```

Les deux autres fonctions permettant d'écrire dans un descripteur sont `writev()` et `pwrite()`. L'appel-système `writev()` est symétrique à `readv()` ; il permet d'écrire une succession de valeurs en un seul appel. Son prototype est défini dans `<sys/uio.h>`. Les conditions d'échec sont les mêmes que celles de `write()`.

Pour que `pwrite()` soit déclaré dans `<unistd.h>`, il faut définir la constante symbolique `_XOPEN_SOURCE` et lui donner la valeur `500` avant l'inclusion du fichier d'en-tête. Cet appel-système fonctionne comme `write()` mais en effectuant l'écriture à la position indiquée en dernier argument et sans modifier la position courante du descripteur. En plus des conditions d'échec identiques à celles de `write()` s'ajoutent celles de l'appel-système de positionnement que nous allons voir à présent. Comme son homologue `pread()`, cet appel-système sert surtout à implémenter les entrées-sorties asynchrones.

Positionnement dans un descripteur de fichier

Il n'existe qu'un seul appel-système, nommé **`lseek()`**, permettant de consulter ou de modifier la position courante dans un descripteur de fichier. Son prototype est déclaré dans `<unistd.h>` :

```
off_t lseek (int descripteur, off_t position, int debut);
```

Le type `off_t` est défini dans `<sys/types.h>` et correspond sur la plupart des systèmes à un `long int`.

Cette fonction permet de déplacer la position courante dans le descripteur à la nouvelle valeur indiquée en second argument. Le point de départ, fourni en troisième argument, peut prendre comme avec `fseek()` l'une des valeurs suivantes : `SEEK_SET`, `SEEK_CUR` ou `SEEK_END`. Cet appel-système renvoie la nouvelle position, mesurée en octets depuis le début du fichier, ou –1 en cas d'erreur. Pour connaître la position courante, il suffit donc d'utiliser `lseek (fd, 0, SEEK_CUR)`.

Nous avons déjà indiqué que le positionnement dans un descripteur est mémorisé dans la table des fichiers et non dans la table des descripteurs. Si un processus ouvre un descripteur avant d'invoquer `fork()`, il partagera avec son fils la structure `file` de la table des fichiers. Le positionnement sera donc commun aux deux processus, tel que l'indique l'exemple ci-après, dans lequel nous avons supprimé toutes les vérifications d'erreur en retour de `lseek()`, afin de simplifier le listing.

exemple_lseek.c :

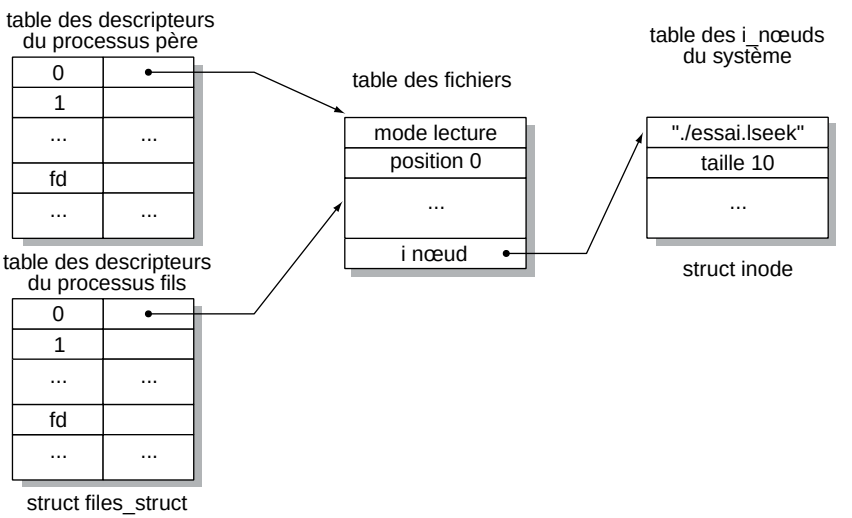
```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <sys/wait.h>

    int
main (void)
{
    int   fd;
    pid_t pid_fils;
    off_t position;

    fd = open("essai.lseek", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    /* On écrit quelques octets */
    if (write(fd, "ABCDEFGHIJ", 10) != 10) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    /* Puis on sépare les processus */
    if ((pid_fils = fork()) < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid_fils) {
        /* Processus père */
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Père : position = %ld \n", position);
        sleep(1);
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Père : position = %ld \n", position);
        lseek(fd, 5, SEEK_SET);
        fprintf(stderr, "Père : déplacement en position 5\n");
        waitpid(pid_fils, NULL, 0);
    } else {
        /* Processus fils */
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Fils : position = %ld \n", position);
        lseek(fd, 2, SEEK_SET);
        fprintf(stderr, "Fils : déplacement en position 2\n");
        sleep(2);
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Fils : position = %ld \n", position);
    }
```

```
    close(fd);
    return EXIT_SUCCESS;
}
```

Dans cet exemple le processus père et le fils modifient alternativement la position d'un descripteur. Nous voyons que les déplacements effectués dans un processus sont immédiatement répercutés dans l'autre :

```
$ ./exemple_lseek
Père : position = 10
Fils : position = 10
Fils : déplacement en position 2
Père : position = 2
Père : déplacement en position 5
Fils : position = 5
$
```

Figure 19.3
Partage de fichier entre processus père et fils



Il est important de remarquer que l'utilisation de `lseek()` n'implique aucune entrée-sortie sur le système de fichier correspondant au descripteur. Il ne s'agit que de la consultation ou de la modification d'un champ de la structure `file`, mais pas d'un accès réel au fichier. L'emploi de `lseek()` n'est donc pas exigeant en termes de performances (mis à part le coût d'un appel-système) ; il ne risque pas de bloquer, mais ne fournit pas non plus de réelle information sur l'état du fichier correspondant. Les erreurs devront être détectées dans les appels-système `read()`, `write()` ou `close()` suivants.

Manipulation et duplication de descripteurs

Nous avons observé qu'en cas d'utilisation de `fork()`, la table des descripteurs correspondant au processus père est copiée dans l'environnement du processus fils, mais que la structure `file` est commune aux deux processus.

Un processus peut aussi employer les appels-système **dup()** ou **dup2()** pour obtenir une seconde copie d'un descripteur ouvert, pointant sur la même structure `file` que l'original. L'intérêt principal de ce mécanisme est de pouvoir modifier les descripteurs d'entrée et de sortie standard, en utilisant des sockets ou des tubes par exemple. Les prototypes de ces appels-système sont déclarés dans `<unistd.h>` :

```
int dup (int descripteur);
int dup2 (int descripteur, int nouveau);
```

La fonction `dup()` permet d'obtenir une copie du descripteur fourni en argument. Cet appel-système garantit que le numéro renvoyé sera le premier disponible dans la table des descripteurs du processus. Nous savons par ailleurs que par tradition les numéros de descripteur de `stdin`, `stdout` et `stderr` sont les trois premiers de cette table. Ainsi nous pouvons rediriger la sortie standard, par exemple dans un fichier, en utilisant le code suivant :

```
fd = open(fichier, ...)
...
close(STDOUT_FILENO);
dup(fd);
```

Le premier numéro libre sera celui du descripteur de `stdout` une fois que celui-ci sera refermé. L'appel `dup()` permettra donc de le réaffecter. Nous avons déjà vu ce genre de comportement avec `freopen()` et les flux, mais l'avantage de `dup()` est de permettre la redirection vers des sockets, des tubes, des files, bref tous les éléments utilisables par le noyau sous forme de descripteurs.

Nous allons dans l'exemple suivant nous contenter d'un `dup()` sur un descripteur de fichier, qu'on met en place sur `stdout`, avant d'invoquer `ls`. L'affichage de ce dernier programme est donc redirigé vers le fichier désiré.

exemple_dup.c

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

    int
main (void)
{
    int fd;
    fd = open("essai.dup", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    close(STDOUT_FILENO);
    if (dup(fd) < 0) {
        perror("dup");
        exit(EXIT_FAILURE);
    }
    close(fd);
    execlp("ls", "ls", NULL);
```

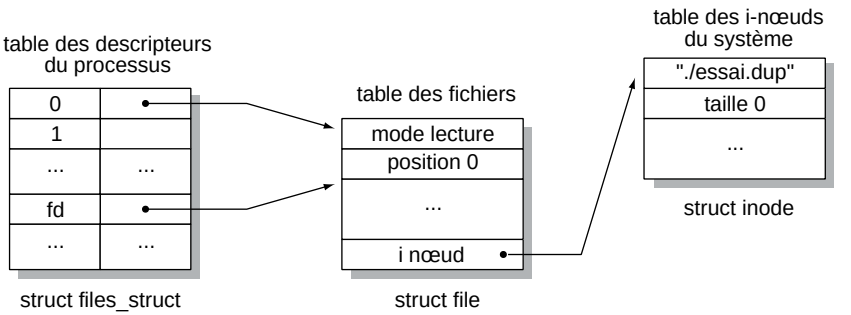
```
    perror("execlp");
    exit(EXIT_FAILURE);
}
```

Nous voyons que la redirection a effectivement lieu :

```
$ ./exemple_dup
$ cat essai.dup
Makefile
cree_core.c
essai.dup
exemple_buffers.c
exemple_dup
exemple_dup.c
exemple_enum.c
exemple_fopen.c
exemple_freopen.c
exemple_fseeko.c
exemple_fseeko_2.c
exemple_ftell.c
exemple_fwrite.c
exemple_lseek.c
exemple_mon_write.c
exemple_open.c
exemple_open_2.c
exemple_open_3.c
exemple_write.c
$
```

Figure 19.4

Duplication d'un descripteur vers la sortie standard



Le défaut de ce procédé réside dans le risque qu'un signal interrompe le processus entre la fermeture du descripteur à rediriger et la duplication du descripteur cible. Si le gestionnaire de ce signal utilise un appel-système `open()`, `creat()`, `pipe()` ou `socket()` par exemple, il va occuper la place qu'on réservait pour la redirection. Aussi le noyau met-il à notre disposition un appel-système `dup2()` qui effectue la redirection complète de manière atomique.

L'invocation de :

```
dup2 (fd, ancien);
```

permet de fermer le descripteur `ancien` s'il est ouvert, puis de dupliquer `fd` en lui associant une nouvelle entrée à la position `ancien` dans la table des descripteurs. Cette méthode est donc plus fiable en ce qui concerne le risque d'interruption par un signal, mais elle permet aussi de rediriger à coup sûr le descripteur voulu, sans présumer de la numérotation effective des descripteurs de `stdin`, `stdout` et `stderr`.

Les programmes offrant des services réseau peuvent choisir d'utiliser leur propre système de connexion, construisant une socket et restant à l'écoute des demandes des clients, ou d'employer les services du démon `inetd`, souvent surnommé le superserveur réseau. Celui-ci gère automatiquement la mise en place d'un serveur sur un port précisé dans le fichier de configuration `/etc/inetd.conf`. Lorsqu'un client établit une connexion, `inetd` se duplique en utilisant `fork()` afin de relancer l'écoute en attente d'un autre client. Le processus fils redirige, avec l'appel-système `dup2()`, son entrée et sa sortie standard vers la socket obtenue, avant de faire appel à `exec()` pour lancer l'application prévue. Celle-ci peut alors travailler directement `stdin` et `stdout` sans se soucier des détails de la programmation réseau.

Les appels-système `dup()`, comme `dup2()`, renvoient le nouveau descripteur obtenu, ou –1 en cas d'erreur. Il existe une différence entre la copie du descripteur et l'original. La table des descripteurs contient en effet un attribut supplémentaire qui est remis à zéro lors de la duplication : l'attribut **close-on-exec**. Lorsqu'un processus invoque un appel-système de la famille `exec()` pour lancer un autre programme, les descripteurs pour lesquels cet attribut est validé sont automatiquement fermés. L'attribut *close-on-exec* est remis à zéro de façon automatique lors d'une duplication, ce qui nous arrange puisqu'on utilise généralement `dup()` ou `dup2()` pour transmettre un fichier ouvert à un processus qu'on veut exécuter.

La modification de l'attribut *close-on-exec* peut se faire, entre autres, à l'aide de l'appel-système **`fcntl()`** qui permet de consulter ou de paramétrer plusieurs aspects d'un descripteur. Cette fonction est déclarée dans `<fcntl.h>` ainsi :

```
int fcntl (int descripteur, int commande, ...);
```

Les points de suspension finals indiquent que des arguments supplémentaires peuvent être ajoutés, en fonction de la commande invoquée. Les commandes disponibles sont variées :

Duplication de descripteur

Avec la commande **`F_DUPFD`**, `fcntl()` permet de dupliquer un descripteur à la manière de `dup()` ou de `dup2()`. Cette commande prend en troisième argument un numéro. Elle duplique le descripteur transmis en premier argument et lui attribue le premier emplacement libre de la table des descripteurs qui soit supérieur ou égal au numéro passé en troisième argument.

Ainsi :

```
fcntl (fd, F_DUPFP, 0);
```

est équivalent à :

```
dup (fd);
```

car il recherche le plus petit descripteur libre. De même :

```
close (ancien);
fcntl (fd, F_DUPFD, ancien);
```

est équivalent à :

```
dup2 (fd, ancien);
```

sauf que `dup2()` renvoie `EBADF` si `ancien` n'est pas dans les valeurs correctes pour un descripteur, alors que `fcntl()` renvoie `EINVAL`.

Accès aux attributs du descripteur

Les commandes **`F_GETFD`** et **`F_SETFD`** permettent de consulter ou de modifier les attributs du descripteur de fichier. De manière portable il n'existe qu'un seul attribut, *close-on-exec*, qu'on représente par la constante **`FD_CLOEXEC`**. Cet attribut est effacé par défaut lors de l'ouverture d'un descripteur.

On peut activer l'attribut *close-on-exec* d'un descripteur en utilisant :

```
etat = fcntl (fd, F_GETFD);
etat |= FD_CLOEXEC;
fcntl (fd, F_SETFD, etat);
```

Le programme ci-dessous ouvre un descripteur de fichier puis, en fonction de son argument en ligne de commande, bascule l'attribut *close-on-exec* du descripteur. Ensuite on invoque l'utilitaire `fuser` en lui indiquant le nom du fichier ouvert. Cette application permet de connaître le PID du ou des processus ayant ouvert le fichier.

exemple_fcntl.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <sys/types.h>

    int
main (int argc, char * argv[])
{
    int  fd;
    int  etat;

    if ((argc != 2)
     || ((strcasecmp(argv[1], "ferme") != 0)
      && (strcasecmp(argv[1], "laisse") != 0))) {
        fprintf(stderr, "syntaxe : %s [ferme | laisse]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fd = open("essai.fcntl", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    if ((etat = fcntl(fd, F_GETFD)) < 0) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }
```

```
    if (strcasecmp(argv[1] , "ferme") == 0)
        etat |= FD_CLOEXEC;
    else
        etat &= ~FD_CLOEXEC;
    if (fcntl(fd, F_SETFD, etat) < 0) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }
    execlp("fuser", "fuser", "-a", "essai.fcntl", NULL);
    perror("execlp");
    exit(EXIT_FAILURE);
}
```

Lorsqu'on exécute le programme avec l'argument « `ferme` », celui-ci active l'attribut *close-on-exec* du descripteur, ce qui déclenchera la fermeture automatique avant d'invoquer `fuser`. Ce dernier nous signale donc qu'aucun processus n'a ouvert le fichier. Par contre, si on fournit l'argument `laisse`, l'attribut est effacé (son état par défaut en fait), et le fichier ne sera donc pas fermé avant d'exécuter `fuser`. Celui-ci détectera alors que le fichier est ouvert et affichera son propre PID.

```
$ ./exemple_fcntl ferme
essai.fcntl:
No process references; use -v for the complete list
$ ./exemple_fcntl laisse
essai.fcntl:          4835
$
```

Pour que cet exemple se déroule correctement, il faut que l'utilitaire `fuser` soit dans le chemin de recherche de `execlp()`. Ceci nécessite éventuellement de rajouter les répertoires `/sbin` ou `/usr/sbin` (où se trouve généralement `fuser`) dans la variable d'environnement `PATH` de l'utilisateur.

Attributs du fichier

Les attributs auxquels on peut accéder avec les commandes **`F_GETFL`** et **`F_SETFL`** sont ceux qui ont été indiqués lors de l'ouverture du fichier avec `open()`. Ces attributs appartiennent à la structure `file` de la table des fichiers et sont donc communs aux différents descripteurs qui pointent sur elle et qui sont obtenus à travers des appels `dup()` ou `fork()`.

Pour consulter le mode d'ouverture d'un fichier, il faut passer la valeur renvoyée à travers le masque **`O_ACCMODE`**, qui permet d'isoler les bits correspondant à `O_RDWR`, `O_RDONLY`, `O_WRONLY`. Le programme suivant permet d'examiner ces modes.

exemple_fcntl_2.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <unistd.h>

    int
main (void)
```

```
{
    int  etat;
    etat = fcntl(STDIN_FILENO, F_GETFL) & O_ACCMODE;
    fprintf(stderr, "stdin : %s\n",
        (etat == O_RDWR) ? "R/W" : (etat == O_RDONLY) ? "R" : "W");
    etat = fcntl(STDOUT_FILENO, F_GETFL) & O_ACCMODE;
    fprintf(stderr, "stdout : %s\n",
        (etat == O_RDWR) ? "R/W" : (etat == O_RDONLY) ? "R" : "W");
    etat = fcntl(STDERR_FILENO, F_GETFL) & O_ACCMODE;
    fprintf(stderr, "stderr : %s\n",
        (etat == O_RDWR) ? "R/W" : (etat == O_RDONLY) ? "R" : "W");
    return EXIT_SUCCESS;
}
```

Il est amusant de voir que le shell configure différemment les descripteurs des flux d'entrée-sortie standard en fonction du fichier sous-jacent :

```
$ ./exemple_fcntl_2
stdin : R/W
stdout : R/W
stderr : R/W
$ ./exemple_fcntl_2 < essai.fcntl
stdin : R
stdout : R/W
stderr : R/W
$ ./exemple_fcntl_2 > essai.fcntl
stdin : R/W
stdout : W
stderr : R/W
$
```

La commande `F_SETFL` ne permet de modifier que les autres éléments du mode d'ouverture : `O_APPEND` et `O_NONBLOCK`.

Attention

Il est recommandé d'utiliser d'abord la commande `F_GETFL` afin d'obtenir l'état complet, puis d'y ajouter ou d'en extraire les constantes désirées avant d'invoquer la commande `F_SETFL`, contrairement à ce qui se fait trop souvent.

L'intérêt de cette commande de modification concerne essentiellement les descripteurs qu'on obtient autrement qu'avec l'appel-système `open()`. Nous y reviendrons donc plus en détail dans le chapitre concernant les communications entre processus.

Quatre autres commandes seront étudiées avec les mécanismes d'entrées-sorties asynchrones puisqu'elles servent à configurer le ou les processus qui sont avertis par un signal lorsque des données sont prêtes à être lues :

- Les commandes `F_GETOWN` et `F_SETOWN` indiquent les processus concernés.
- Les commandes `F_GETSIG` et `F_SETSIG` précisent le signal à utiliser.

Verrouillage d'un descripteur

On peut verrouiller l'accès à un fichier pour en assurer l'exclusivité de deux manières : avec la fonction `flock()` et avec les commandes **`F_GETLK`**, **`F_SETLK`** ou **`F_SETLKW`** de `fcntl()`. Ces deux méthodes sont distinctes et n'ont pas de répercussions l'une sur l'autre. La commande `flock()` est un héritage BSD qu'il vaut mieux éviter de nos jours.

Il est possible, avec `fcntl()`, de verrouiller une partie d'un fichier afin de garantir qu'un seul processus à la fois pourra modifier cette portion. Ce verrouillage peut être **coopératif**, ce qui signifie que les processus doivent vérifier eux-mêmes l'existence d'un verrou et s'abstenir de faire des modifications s'ils en trouvent un. C'est le seul comportement véritablement défini par Posix. Le défaut de ce mécanisme est l'impossibilité de se prémunir des modifications sauvages effectuées par un processus ne se pliant pas à l'autorité du verrouillage. Pour cela, le noyau Linux implémente comme de nombreux autres systèmes Unix un verrouillage **strict**. La portion de fichier ainsi bloquée est totalement immunisée contre les modifications par d'autres processus, même s'ils sont exécutés par *root*. La distinction entre verrouillage coopératif et verrouillage strict se fait au niveau du fichier lui-même, aussi étudierons-nous d'abord le principe du verrou coopératif, puis nous verrons comment le transformer en verrou strict.

Les verrous sont représentés par des structures **`flock`** qui sont définies dans `<fcntl.h>`, avec les cinq membres suivants :

Nom	Type	Utilisation
`l_type`	`short int`	Ce membre indique le type de verrouillage. Il peut s'agir de `F_RDLCK` pour un verrou en lecture, `F_WRLCK` pour un verrou en écriture, ou `F_UNLCK` pour supprimer le verrou.
`l_whence`	`short int`	On signale ainsi le point de départ de la mesure annonçant le début du verrouillage. I C'est l'équivalent du troisième argument de `lseek()`, qui peut prendre les valeurs `SEEK_SET`, `SEEK_CUR` ou `SEEK_END`.
`l_start`	`off_t`	Ce champ précise le début de la portion verrouillée du fichier.
`l_len`	`off_t`	Longueur de la partie à verrouiller dans le fichier, mesurée en octets.
`l_pid`	`pid_t`	Ce membre est rempli automatiquement par le système pour indiquer le processus détenteur d'un verrou.

Le type de verrou, indiqué dans le premier membre de cette structure, a la signification suivante :

- `F_RDLCK` : le processus demande un accès en lecture sur la portion concernée du fichier, en s'assurant ainsi qu'aucun autre processus ne viendra modifier la partie qu'il lit. Plusieurs processus peuvent disposer simultanément d'un verrouillage en lecture sur la même portion de fichier.
- `F_WRLCK` : le processus veut modifier une partie du fichier. Il s'assure ainsi qu'aucun autre processus ne risque d'écrire au même endroit mais également qu'aucun ne tentera de verrouiller en lecture la portion concernée.

Le comportement peut donc être résumé ainsi :

- Si une zone d'un fichier n'a aucun verrou, un processus pourra en placer un en lecture ou en écriture.

- Si une zone est verrouillée en lecture, un autre verrou en lecture sera accepté, mais pas un verrou en écriture.
- Si une zone dispose d'un verrou en écriture, aucun autre verrouillage ne sera accepté.

Lorsqu'on parle de deux verrouillages sur la même zone, il suffit en fait que les deux zones verrouillées aient une intersection non vide. Le noyau vérifie en effet les superpositions des portions demandées.

Si on indique une longueur *l_len* nulle, cela signifie « jusqu'à la fin du fichier ». Bien entendu, le point de départ peut être placé n'importe où, éventuellement au début si on veut verrouiller tout le fichier. Le verrouillage peut s'étendre au-delà de la fin du fichier si on désire y inscrire de nouvelles données.

Pour placer un verrou sur une portion d'un fichier, on peut employer les commandes `F_SETLK` ou `F_SETLKW` de `fcntl()`. Cette dernière commande est bloquante (`W` signifie *wait*). L'appel-système `fcntl()` reste bloqué dans ce cas si un verrou est déjà présent, jusqu'à ce qu'il soit retiré. Cette fonction est toutefois interruptible par un signal, dans ce cas elle échoue et renvoie `EINTR`. La commande `F_SETLK` ne reste pas bloquée mais peut renvoyer `EACCES` ou `EAGAIN` suivant le type de verrouillage déjà présent. Voici donc deux méthodes de verrouillage en écriture de l'ensemble du fichier.

```
struct flock  lock;
char          chaine[2];

lock.l_type = F_WRLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;

while (fcntl(fd, F_SETLK, & lock) < 0) {
    fprintf(stdout, "Fichier verrouillé, réessayer ? ");
    fgets(chaine, 2, stdin);
    if (toupper(chaine[0]) == 'O')
        continue;
    return -1;
}
/* Ici l'accès est autorisé,         */
/* on peut faire les modifications,  */
/* puis libérer le verrou            */
fcntl(fd, F_UNLCK, & lock);
return 0;
```

Voici à présent l'attente bloquante :

```
struct flock lock;
char   chaine[2];

lock.l_type = F_WRLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;

while (fcntl(fd, F_SETLKW, & lock) < 0)
    if (errno != EINTR)
```

```
        return -1;
/* Ici l'accès est autorisé,         */
/* on peut faire les modifications,  */
/* puis libérer le verrou            */
fcntl(fd, F_UNLCK, & lock);
return 0;
```

On peut aussi demander l'état du verrouillage sur un fichier en utilisant la commande `F_GETLK`, le troisième argument étant un pointeur sur une structure `flock`, comme avec `F_SETLK`. Cette structure sera modifiée au cours de l'appel pour représenter le verrou actuellement actif qui bloque l'accès à la portion voulue. Si aucun verrou n'est présent, le membre `l_type` est alors rempli avec la valeur `F_UNLCK`. Cette commande ne doit être utilisée qu'avec précaution, car l'état du fichier peut très bien être modifié entre le retour de l'appel-système `fcntl()` et l'instruction suivante. Il ne faut s'en servir qu'à titre indicatif, notamment pour connaître le PID du processus tenant le fichier, comme dans cette attente non bloquante :

```
struct flock actuel;

...

while (fcntl(fd, F_SETLK, & lock) < 0) {
    /* Copier le verrou voulu dans la structure servant */
    /* pour l'interrogation                             */
    memcpy(& actuel, & lock, sizeof (struct flock));
    /* Interroger le noyau sur le verrouillage          */
    if (fcntl(fd, F_GETLK, & actuel) < 0)
        continue;
    if (actuel.l_type == F_UNLCK)
        /* Le verrou a été supprimé entre temps      */
        continue;
    fprintf(stdout, "Fichier verrouillé par processus %d, réessayer ?",
                    actuel.l_pid);
    fgets(chaine, 2, stdin);
    if (toupper(chaine[0]) == 'O')
        continue;
    return -1;
}
...
```

Il existe des situations où le verrouillage d'un descripteur conduit à un interblocage de deux processus. Supposons en effet que chaque processus a verrouillé une partie d'un fichier et réclame chacun un second verrou sur la partie tenue par l'autre. On arrive à une situation de blocage « à mort » que le noyau doit détecter et essayer d'éviter. Ceci peut se produire notamment quand plusieurs copies d'un même processus tentent simultanément d'ajouter des données à la fin d'un fichier et de mettre à jour une table des matières située au début. Dans un tel cas, le noyau fait échouer la tentative de verrouillage avec l'erreur `EDEADLK` (*dead lock*), comme nous allons le voir avec l'exemple suivant.

exemple_fcntl_3.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
```

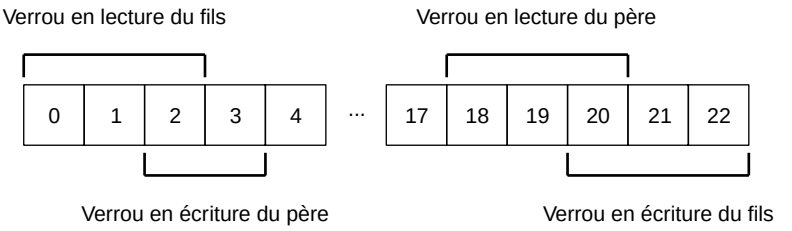
```
    #include <unistd.h>
    #include <sys/wait.h>

    int
main (void)
{
    int          fd;
    pid_t        pid;
    struct flock lock;

    /* Création d'un fichier avec quelques données */
    fd = open("essai.fcntl", O_RDWR | O_CREAT | O_TRUNC, 0644)
    if (fd < 0){
        perror("open");
        exit(EXIT_FAILURE);
    }
    write(fd, "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 26);
    if ((pid = fork()) == 0) {
        fprintf(stderr, "FILS : Verrou en Lecture de 0-1-2\n");
        lock.l_type = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 0;
        lock.l_len = 3;
        if (fcntl(fd, F_SETLKW, & lock) < 0)
            perror("FILS");
        else
            fprintf(stderr, "FILS : Ok\n");
        sleep(1);
        fprintf(stderr, "FILS : Verrou en Ecriture de 20-21-22\n");
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 20;
        lock.l_len = 3;
        if (fcntl(fd, F_SETLKW, & lock) < 0)
            perror("FILS");
        else
            fprintf(stderr, "FILS : Ok\n");
        sleep(2);
    } else {
        fprintf(stderr, "PERE : Verrou en Lecture de 18-19-20\n");
        lock.l_type = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 18;
        lock.l_len = 3;
        if (fcntl(fd, F_SETLKW, & lock) < 0)
            perror("PERE");
        else
            fprintf(stderr, "PERE : Ok\n");
        sleep(2);
        fprintf(stderr, "PERE : Verrou en Ecriture de 2-3\n");
        lock.l_type = F_WRLCK;
        lock.l_whence = SEEK_SET;
```

```
        lock.l_start = 2;
        lock.l_len = 2;
        if (fcntl(fd, F_SETLKW, & lock) < 0)
            perror("PERE");
        else
            fprintf(stderr, "PERE : Ok\n");
        fprintf(stderr, "PERE : Libération du verrou 18-19-20\n");
        lock.l_type = F_UNLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 18;
        lock.l_len = 3;
        if (fcntl(fd, F_SETLKW, & lock) < 0)
            perror("PERE");
        else
            fprintf(stderr, "PERE : Ok\n");
        waitpid(pid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

Nous remarquons que les zones verrouillées par les deux processus ne coïncident pas tout à fait, elles ont simplement des intersections communes.

Figure 19.5

Blocage entre père et fils



L'exécution montre bien que le noyau détecte un risque de blocage complet et fait échouer un appel-système `fcntl()` :

```
$ ./exemple_fcntl_3
PERE : Verrou en Lecture de 18-19-20
FILS : Verrou en Lecture de 0-1-2
FILS : Ok
PERE : Ok
FILS : Verrou en Ecriture de 20-21-22
PERE : Verrou en Ecriture de 2-3
PERE: Blocage évité des accès aux ressources
PERE : Libération du verrou 18-19-20
PERE : Ok
FILS : Ok
$
```

Encore une fois, nous nous sommes contenté d'utiliser des sommeils `sleep()` pour synchroniser les différentes phases des processus, ce qui ne fonctionne véritablement que sur un système peu chargé, mais permet de conserver des exemples assez simples. La détection des situations de blocage complet est assez performante puisqu'elle marche également quand de multiples processus tiennent chacun un maillon d'une chaîne en attendant la libération du suivant. Nous allons le démontrer en implémentant de manière simplifiée le fameux *repas des philosophes*, présenté par Dijkstra. Nous asseyons *n* philosophes autour d'une table, chacun ayant une assiette de spaghettis devant lui. Il y a *n* fourchettes sur la table, une entre chaque assiette, et on considère que pour manger des spaghettis, il faut deux fourchettes. Plusieurs difficultés peuvent être mises en relief avec ce problème classique, mais nous allons simplement montrer une situation de blocage où chaque philosophe prend la fourchette à gauche de son assiette, puis attend que la fourchette de droite soit libre. Bien sûr, ils restent tous en attente si le noyau ne détecte pas le blocage.

exemple_fcntl_4.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/wait.h>

    void    philosophe (int numero, int total, int fd);

    int
main (int argc, char * argv[])
{
    int  n;
    int  i;
    int fd;

    if ((argc != 2) || (sscanf(argv[1], "%d", & n) != 1)) {
        fprintf (stderr, "Syntaxe : %s nb_philosophes\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fd = open("essai.fcntl", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0){
        perror("open");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < n; i ++)
        write(fd, "X", 1);
    for (i = 0; i < n; i ++) {
        if (fork() != 0)
            continue;
        philosophe(i, n, fd);
        exit(EXIT_SUCCESS);
    }
    for (i = 0; i < n; i ++)
        wait(NULL);
    return EXIT_SUCCESS;
}
```

```
    void
philosophe (int numero, int total, int fd)
{
    struct flock lock;
    char   nom[10];

    sprintf(nom, "FILS %d", numero);

    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_len = 1;
    lock.l_start = numero;
    fprintf(stderr, "%s : fourchette gauche (%ld)\n",
                 nom, lock . l_start);
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror(nom);
    else
        fprintf(stderr, "%s : Ok\n", nom);

    sleep(1);
    lock.l_start = (numero + 1) % total;
    fprintf(stderr, "%s : fourchette droite (%ld)\n",
                    nom, lock . l_start);
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror(nom);
    else
        fprintf(stderr, "%s : Ok\n", nom);

    sleep(1);
    lock.l_type = F_UNLCK;
    fprintf(stderr, "%s : repose fourchette (%ld)\n",
                    nom, lock . l_start);
    fcntl(fd, F_SETLKW, & lock);
    lock.l_start = numero;
    fprintf(stderr, "%s : repose fourchette (%ld)\n",
                    nom, lock . l_start);
    fcntl(fd, F_SETLKW, & lock);
}
```

On remarquera au passage que le blocage peut être évité en ne prenant pas systématiquement les fourchettes dans l'ordre gauche-droite, mais en prenant celles de rang pair en premier, puis celles de rang impair. Ici, le blocage est bien détecté par le noyau :

```
$ ./exemple_fcntl_4
Syntaxe : ./exemple_fcntl_4 nb_philosophes
$ ./exemple_fcntl_4 4
FILS 0 : fourchette gauche (0)
FILS 0 : Ok
FILS 1 : fourchette gauche (1)
FILS 1 : Ok
FILS 2 : fourchette gauche (2)
FILS 2 : Ok
FILS 3 : fourchette gauche (3)
```

```
FILS 3 : Ok
FILS 0 : fourchette droite (1)
FILS 1 : fourchette droite (2)
FILS 2 : fourchette droite (3)
FILS 3 : fourchette droite (0)
FILS 3: Blocage évité des accès aux ressources
FILS 3 : repose fourchette (0)
FILS 3 : repose fourchette (3)
FILS 2 : Ok
FILS 2 : repose fourchette (3)
FILS 2 : repose fourchette (2)
FILS 1 : Ok
FILS 1 : repose fourchette (2)
FILS 1 : repose fourchette (1)
FILS 0 : Ok
FILS 0 : repose fourchette (1)
FILS 0 : repose fourchette (0)
$
```

Les verrouillages que nous avons vus jusqu'à présent sont de type coopératif, ce qui signifie que chaque processus désireux de modifier un fichier doit se discipliner et utiliser les procédures d'accès adéquates. Aucune protection n'est assurée contre un processus qui outrepasse les verrouillages et modifie le fichier de manière anarchique. Pour éviter cela, le noyau implémente un mécanisme de verrouillage strict. Il suffit simplement de modifier le mode de protection du fichier et tous les verrouillages vus précédemment seront automatiquement renforcés par le noyau.

Un fichier est marqué comme verrouillable de manière stricte en modifiant les bits d'autorisation pour positionner le bit Set-GID tout en effaçant la permission d'exécution pour le groupe. Cette combinaison n'a pas de sens par ailleurs, aussi a-t-elle été choisie comme marque de verrouillage strict. On peut fixer les bits voulus ainsi :

```
$ chmod g-x fichier
$ chmod g+s fichier
```

ou à l'ouverture du fichier

```
fd = open(fichier, O_RDWR | O_CREAT | O_EXCL, 02644);
```

Attention

Certains systèmes de fichiers, par exemple *msdos* ou *vfat*, ne permettent pas de fixer les attributs Set-GID des fichiers. Les verrous y seront donc toujours coopératifs.

Lorsqu'un verrouillage est ainsi placé sur une portion de fichier, toutes les tentatives de modification de son contenu échoueront. Dans l'exemple suivant nous créons un fichier que nous verrouillons entièrement, puis nous attendons que l'utilisateur appuie sur Entrée pour le libérer.

exemple_fcntl_5.c :

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
```

```
    #include <unistd.h>

    int
main (int argc, char * argv[])
{
    char    chaine[80];
    int             fd;
    struct flock flock;
    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_fichier \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fd = open(argv[1], O_RDWR | O_CREAT | O_EXCL, 02644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    write(fd, "ABCDEFGHIJ", 10);
    flock.l_type = F_WRLCK;
    flock.l_start = 0;
    flock.l_whence = SEEK_SET;
    flock.l_len = 10;
    if (fcntl(fd, F_SETLK, & flock) < 0) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Verrou installé \n");
    fgets(chaine, 80, stdin);
    close(fd);
    return EXIT_SUCCESS;
}
```

Le blocage strict peut parfois être dangereux, principalement avec des répertoires exportés en NFS, car même *root* ne peut plus modifier le contenu d'un fichier verrouillé par un utilisateur. Aussi, le noyau gère-t-il une validation partition par partition des verrouillages stricts. En d'autres termes, il faut que la partition contenant le système de fichiers considéré ait été montée avec l'option `mand`, qui autorise les *mandatory locks*, c'est-à-dire les verrous stricts. Cette option n'est généralement pas validée par défaut, aussi *root* doit-il modifier le fichier `/etc/fstab` pour ajouter l'option `mand` et remonter la partition (ce qui ne nécessite pas de redémarrer la machine pour autant).

```
$ su
Password:
#       ...modification de /etc/fstab...
# cat /etc/fstab
/dev/hda5    /     ext3    defaults,mand   1 1
  [...]
# mount -o remount /
# logout
$ rm essai.*
$ ./exemple_fcntl_5 essai.write
Verrou installé
```

Nous exécutons alors sur un autre terminal les tentatives d'écriture suivantes :

```
$ ls -l essai.write
-rw-r-Sr--   1 ccb    ccb    10 Nov 19 16:12 essai.write
$ cat exemple_fcntl_5.c > essai.write
bash: essai.write: Ressource temporairement non disponible
$ su
Password:
# cat exemple_fcntl_5.c > essai.write
bash: essai.write: Ressource temporairement non disponible
# logout
$
```

Les appels-système `write()` de l'utilitaire `cat` échouent avec l'erreur `EAGAIN`. Notons toutefois que le fichier n'est pas pour autant protégé contre une suppression par `rm`, car c'est l'écriture dans le répertoire qui est concernée lors d'un effacement et non une écriture dans le fichier. Ceci peut poser des problèmes avec les applications qui modifient des fichiers en commençant par en effectuer une copie (par exemple dans `/tmp`) avant de supprimer l'original et de le réécrire.

Autre méthode de verrouillage

Il existe un autre appel-système permettant le verrouillage coopératif de fichiers. Héritée de l'univers BSD, la fonction **`flock()`** n'est pas normalisée par SUSv3 et est de moins en moins utilisée. Elle permet de verrouiller uniquement un fichier complet par l'intermédiaire d'un descripteur. Le prototype est déclaré dans `<sys/file.h>` :

```
int flock (int descripteur, int commande);
```

Les commandes possibles sont les suivantes :

- **`LOCK_SH`** : pour placer un verrou partagé. Plusieurs processus peuvent disposer simultanément d'un verrou partagé sur le même fichier. C'est l'équivalent des verrous `F_RDLCK` de `fcntl()`, qui permettent de s'assurer que le fichier ne sera pas modifié pendant qu'on veut le lire.
- **`LOCK_EX`** : pour placer un verrou exclusif. Il ne peut y avoir qu'un seul verrouillage exclusif à la fois. Un processus l'utilise lorsqu'il veut écrire dans le descripteur.
- **`LOCK_UN`** : pour supprimer un verrouillage précédemment installé.

On peut également ajouter avec un OU binaire la constante `LOCK_NB` pour empêcher la demande de verrouillage d'être bloquante. Si un verrou incompatible est déjà présent, l'appel-système `flock()` échouera en renvoyant –1 et en plaçant `EWOULDBLOCK` dans `errno`. Lorsqu'il réussit, l'appel `flock()` renvoie 0.

Le verrouillage avec `flock()` n'est **pas** compatible avec celui qui est fourni par `fcntl()`, et les verrous `flock()` ne sont jamais renforcés de manière stricte par le noyau. De plus, il ne permet pas de bloquer uniquement certaines parties d'un fichier. Aussi évitera-t- on au maximum de l'employer.

Conclusion

Nous avons étudié dans ce chapitre l'essentiel des fonctionnalités concernant la manipulation des fichiers, sous forme de descripteurs de bas niveau.

Nous nous intéresserons dans les chapitres à venir aux fichiers en tant qu'entités sur le disque, en étudiant les répertoires, les autorisations d'accès, etc. Nous reviendrons sur certaines fonctionnalités concernant les descripteurs dans le chapitre sur les communications entre processus et dans celui sur la programmation réseau, ainsi que sur tout ce qui a trait aux possibilités d'entrées-sorties asynchrones.

20

Accès au contenu des répertoires

Il est souvent important pour un programme de pouvoir afficher la liste des fichiers contenus dans un répertoire. Ceci ne concerne pas uniquement les utilitaires du genre `ls` ou les gestionnaires de fichiers, mais peut servir à toute application proposant l'enregistrement et la récupération de données.

Les interfaces graphiques actuelles permettent de plus en plus facilement de disposer de boîtes de dialogue pour la sauvegarde ou la lecture de fichiers, sans avoir à écrire manuellement le code de parcours d'un répertoire. Toutefois, dans certaines situations, l'accès au contenu d'un répertoire est indispensable, notamment lorsque le nom d'un fichier constitue une information importante pour analyser son contenu. Je peux citer le cas d'une application recevant des fichiers de données météorologiques, dont le nom permet de retrouver la zone couverte ainsi que l'heure de capture. Ces informations pourraient à profit être intégrées dans une section d'en-tête du fichier, mais on n'est pas toujours responsable du format des données fournies en amont d'un système, principalement dans un environnement hétérogène incluant des dispositifs provenant de divers fournisseurs.

Nous allons donc dans ce chapitre nous concentrer sur l'accès au contenu d'un répertoire, la lecture de la liste des fichiers s'y trouvant, la modification du répertoire de travail, la suppression de sous-répertoires ou de fichiers, ainsi que la recherche de noms de fichiers par des caractères génériques et le parcours récursif, à la manière de l'utilitaire `find`.

Lecture du contenu d'un répertoire

Sous Unix, un répertoire est un fichier spécial, contenant pour chaque fichier ou sous-répertoire une structure opaque variant suivant le type de système de fichier. À titre d'exemple, avec le système ext3, les répertoires comprennent des structures `ext3_dir_entry` définies `dans`

<linux/ext3fs.h>. Chaque structure dispose du nom du fichier, de son numéro d'i-nœud, ainsi que des champs servant à la gestion interne des structures.

Au niveau applicatif, les fonctions **opendir()**, **readdir()**, **closedir()** nous permettent d'accéder au contenu d'un répertoire sous forme de structures **dirent**. Pour assurer la portabilité d'une application, nous nous limiterons à l'utilisation des deux seuls champs qui soient définis par SUSv3, **char d_name[]**, qui contient le nom du fichier ou du sous-répertoire et **ino_t d_ino**, le numéro d'i-nœud du fichier (ce dernier champ est accessible si la constante symbolique _XOPEN_SOURCE contient la valeur 500).

Ces fonctions sont définies dans <dirent.h> :

```
DIR * opendir (const char * repertoire);
struct dirent * readdir (DIR * dir);
int closedir(DIR * dir);
```

Le type DIR, défini dans <sys/types.h>, est une structure opaque, comparable au flux FILE, mais on l'emploie sur des répertoires au lieu des fichiers. À la manière de fopen(), la fonction opendir() renvoie un pointeur NULL en cas d'échec. La fonction readdir() renvoie l'entrée suivante ou NULL une fois arrivée à la fin du répertoire. Lorsqu'on a fini d'utiliser le répertoire, on le referme avec closedir() :

exemple_opendir.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <dirent.h>
    #include <sys/types.h>

    void
affiche_contenu (const char * repertoire)
{
    DIR *           dir;
    struct dirent * entree;

    dir = opendir(repertoire);
    if (dir == NULL)
        return;
    fprintf(stdout, "%s :\n", repertoire);
    while ((entree = readdir(dir)) != NULL)
        fprintf (stdout, "   %s\n", entree->d_name);
    fprintf(stdout, "\n");
    closedir(dir);
}

    int
main (int argc, char * argv[])
{
    int   i;
    if (argc < 2)
        affiche_contenu(".");
    else for (i = 1; i < argc; i ++)
        affiche_contenu(argv[i]);
    return EXIT_SUCCESS;
}
```

Ce programme affiche le contenu des répertoires dont le nom est passé en argument.

```
$ ./exemple_opendir /etc/X11/xdm /proc/tty/
/etc/X11/xdm :
   .
   ..
   GiveConsole
   TakeConsole
   Xaccess
   Xresources
   Xservers
   Xsession
   Xsetup_0
   chooser
   xdm-config
   authdir
   Xsetup_0.rpmsave

/proc/tty/ :
   .
   ..
   drivers
   ldiscs
   driver
   ldisc

$
```

Le pointeur renvoyé par `readdir()` est une variable statique, qui peut être écrasée à chaque appel. Cette fonction n'est donc pas réentrante et ne doit pas être utilisée dans un contexte multithread. Pour cela, on peut employer la fonction **`readdir_r()`** qui prend deux arguments supplémentaires pour stocker la valeur de retour et mémoriser la position suivante dans le répertoire.

```
int readdir_r (DIR * dir,
               struct dirent * entree,
               struct dirent ** memorisation) ;
```

Cette fonction transmet 0. Arrivée à la fin du répertoire, elle renvoie également 0, mais l'argument `memorisation` vaut `NULL`. Voici comment l'utiliser :

```
    struct dirent resultat;
    struct dirent * memorisation ;
    DIR * dir;

    while (1) {
        if (readdir(dir, & resultat, & memorisation) != 0)
            return -1;
        if (memorisation == NULL)
            break;
        fprintf(stdout, "   %s\n", entree->d_name);
    }
    fprintf(stdout, "\n");
```

On peut s'interroger sur l'allocation mémoire nécessaire pour stocker la chaîne de caractères contenant le nom des éléments. En fait, cette chaîne dispose automatiquement d'une taille maximale, définie par la constante **NAME_MAX**. La structure `dirent` comprend aussi sur la plupart des systèmes (mais pas tous) un membre `d_namlen` contenant la longueur du membre `d_name` comme la valeur renvoyée par `strlen()` (donc caractère nul final non compté). Ce champ n'est toutefois pas défini par SUSv3, et on évitera autant que possible de l'employer.

On observe que `readdir()` et `readdir_r()` renvoient les entrées « `.` » et « `..` » correspondant respectivement au répertoire courant et à son parent. Ce comportement n'est pas garanti par SUSv3. Par contre, ces deux entrées sont toujours valables pour `opendir()` ou pour des commandes de changement de répertoire de travail que nous verrons plus loin.

Comme avec les flux de fichiers, il est possible de se déplacer au sein des répertoires `DIR` en utilisant **rewinddir()**, qui revient au début du répertoire, **telldir()**, qui renvoie la position courante, ou **seekdir()**, qui permet de sauter à une position donnée, renvoyée précédemment par `telldir()`. Les prototypes de ces fonctions sont :

```
void rewinddir (DIR * dir);
void seekdir (DIR * dir, off_t offset);
off_t telldir(DIR * dir);
```

Ces trois fonctions sont définies par SUSv3.

Il existe également une fonction puissante permettant de sélectionner une partie du contenu d'un répertoire, de la trier et d'en fournir le contenu dans une table allouée automatiquement. Cette fonction est nommée **scandir()**, et son prototype peut paraître un peu inquiétant au premier coup d'œil :

```
int scandir(const char * dir, struct dirent ***namelist,
            int (* selection) (const struct dirent * entree),
            int (* comparaison) (const struct dirent ** entree_1,
                                 const struct dirent ** entree_2));
```

La fonction `scandir()` commence par lire entièrement le contenu du répertoire dont le nom lui est fourni en premier argument. Ensuite, elle invoque pour chaque entrée du répertoire la fonction `selection` sur laquelle on lui passe un pointeur en troisième argument. Si la fonction `selection()` renvoie une valeur nulle, l'entrée considérée est rejetée. Sinon, elle est sélectionnée.

Puis, `scandir()` trie la table des entrées restantes, en invoquant la routine `qsort()` que nous avons étudiée au chapitre 17. Pour pouvoir trier la table, on utilise comme fonction de comparaison celle dont le pointeur est fourni en dernier argument de `scandir()`.

Une fois la table triée, `scandir()` met à jour le pointeur passé en second argument pour le diriger dessus. Les allocations ayant lieu avec `malloc()`, il faudra libérer ensuite le contenu de cette table.

Si on désire sélectionner tout le contenu du répertoire, il est possible de transmettre un pointeur `NULL` en guise de fonction de sélection. La bibliothèque GlibC met également à notre disposition une fonction **alphasort()** qui permet de trier automatiquement les entrées du répertoire par ordre alphabétique :

```
int alphasort (const struct dirent ** entree_1,
               const struct dirent ** entree_2);
```

Nous allons utiliser `scandir()` et `alphasort()` pour créer un exemple permettant de sélectionner les éléments correspondant à une expression rationnelle dans un répertoire donné. Les fonctions `regcomp()` et `regexec()` traitant les expressions rationnelles ont été présentées dans le chapitre 16.

exemple_scandir.c :

```
    #include <dirent.h>
    #include <regex.h>
    #include <stdio.h>
    #include <stdlib.h>

    regex_t  motif_recherche;

    int
selection(const struct dirent * entree)
{
    if (regexec(& motif_recherche, entree -> d_name, 0, NULL, 0) == 0)
        return 1;
    return 0;
}

    int
main (int argc, char * argv[])
{
    struct dirent ** liste;
    int              nb_entrees;
    int              i;
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s répertoire motif\n", argv [0]);
        exit(EXIT_FAILURE);
    }
    if (regcomp(& motif_recherche, argv[2], REG_NOSUB) !=0) {
        fprintf(stderr, "Motif illégal\n");
        exit(EXIT_FAILURE);
    }
    nb_entrees = scandir(argv[1], & liste, selection, alphasort);
    if (nb_entrees <= 0)
        exit(EXIT_SUCCESS);
    for (i = 0; i < nb_entrees; i ++) {
        fprintf(stdout, "  %s\n", liste[i]->d_name);
        free(liste[i]);
    }
    fprintf(stdout, "\n");
    free(liste);
    return EXIT_SUCCESS;
}
```

Dans l'exécution suivante, on remarque deux choses :

- Les expressions rationnelles ne sont pas identiques aux motifs génériques du shell (principalement en ce qui concerne le métacaractère `*`). Nous verrons plus loin dans ce chapitre des fonctions permettant d'obtenir le même comportement.

- Il ne faut pas oublier de protéger du shell les caractères spéciaux `[`, `]`, `|`…

```
$ ./exemple_scandir /etc/ shad
  shadow
  shadow-
  shadow.bak

$ ./exemple_scandir /usr/bin/ \[a\|b\]cc
  access
  bcc
  byacc
  pgaccess
  yacc

$
```

Un répertoire est, nous l'avons dit, considéré comme un fichier particulier, mais un fichier quand même. Il est donc possible d'ouvrir avec `open()` un descripteur sur ce fichier. L'ouverture ne peut se faire qu'en lecture, car seul le noyau a le droit de modifier le contenu véritable du répertoire, pour être sûr de garder l'ensemble cohérent. Le droit d'écriture sur un répertoire correspond simplement à l'autorisation d'y créer un nouveau fichier – avec `open()` par exemple – ou un sous-répertoire, mais en passant toujours par l'intermédiaire d'un appel-système qui permet au noyau de contrôler les données.

Il n'est pas possible de lire directement le contenu d'un descripteur de répertoire avec `read()`. Il faut utiliser des appels-système compliqués, comme `getdents()` ou la version bas niveau de `readdir()`. Ces appels-système peuvent varier d'une version à l'autre du noyau, et les structures de données qu'ils manipulent ne sont pas portables. Nous ne les présenterons donc pas dans cet ouvrage. Le lecteur ayant véritablement besoin d'employer ces fonctions pourra se reporter aux sources du noyau ou à celles de la bibliothèque C pour y étudier les détails d'implémentation.

Changement de répertoire de travail

Chaque processus dispose en permanence d'un répertoire de travail. Ce répertoire est hérité du processus père et peut être modifié avec l'appel-système `chdir()`. Le changement n'est toutefois visible que dans le processus courant et ses futurs descendants, pas dans le processus père.

Lors de la connexion d'un utilisateur, `login` lit dans le fichier `/etc/password` le répertoire personnel de l'utilisateur et s'y place, avant d'invoquer le shell. Il configure également la variable d'environnement `HOME`, qui restera donc correctement renseignée, même si l'utilisateur se déplace dans l'arborescence du système de fichiers.

Il existe deux appels-système permettant de modifier le répertoire courant d'un processus : **`chdir()`**, qui prend en argument le nom du répertoire destination, et **`fchdir()`**, qui utilise un descripteur sur le répertoire cible. Ces deux appels-système sont déclarés dans `<unistd.h>` et définis par SUSv3.

```
int chdir (const char * nom);
int fchdir(int descripteur);
```

Ils renvoient tous deux 0 en cas de réussite, et –1 en cas d'erreur.

Nous avons indiqué qu'un processus dispose toujours d'un répertoire de travail, mais aussi surprenant que cela puisse paraître, il n'existe pas d'appel système permettant d'obtenir directement le nom de ce répertoire. Il faut pour cela s'adresser à une fonction de bibliothèque comme **getcwd()**, **getwd()**, ou **get_current_working_dir_name()**. Cette dernière fonction n'est pas définie par SUSv3. Les trois prototypes sont déclarés dans `<unistd.h>` :

```
char * getcwd (char * buffer, size_t taille);
char * getwd(char * buffer);
char * get_current_working_dir_name(void);
```

La fonction `getcwd()` copie le chemin du répertoire courant dans le buffer transmis, dont on précise également la taille. Si le buffer n'est pas suffisamment grand, `getcwd()` échoue avec l'erreur `ERANGE`. Nous verrons plus bas le moyen de gérer cette situation. Avec la bibliothèque GlibC, il est possible de transmettre un buffer `NULL`, avec une taille valant 0, pour que `getcwd()` assure elle-même l'allocation mémoire nécessaire. Ce comportement n'est malheureusement pas portable sur d'autres systèmes.

La fonction `get_current_working_dir_name()` est une extension Gnu (requérant donc la définition de la constante `_GNU_SOURCE`). Elle alloue automatiquement la taille requise pour le stockage du chemin d'accès, en appelant `malloc()`. Il faudra donc libérer le pointeur obtenu.

La fonction `getwd()` est un héritage de BSD. Il faut donc définir la constante `_BSD_SOURCE`. Cette fonction suppose que le buffer transmis contient au moins **PATH_MAX** octets. Si ce n'est pas le cas, elle risque de déclencher silencieusement un débordement. Il faut donc éviter à tout prix d'employer cette routine.

Le fonctionnement interne traditionnel de `getcwd()` sous Unix est surprenant. Pour obtenir le nom du répertoire courant, `getcwd()` commence par mémoriser le numéro d'i-nœud de l'entrée « `.` » du répertoire courant. Ensuite, elle analyse toutes les entrées du répertoire « `..` » pour retrouver celle dont le numéro d'i-nœud correspond. Nous verrons au chapitre suivant le moyen d'accéder à cette information. Ensuite, le procédé est répété en remontant jusqu'à la racine du système de fichiers. Il est alors possible de reconstituer le chemin complet.

Toutefois, sous Linux, le noyau met à la disposition de la bibliothèque le pseudo-système de fichiers `/proc`, qui contient des informations diverses sur le système. Il existe un sous-répertoire pour chaque processus en cours (par exemple `/proc/524/`). Dans ce sous-répertoire, on trouve divers fichiers, dont un lien symbolique nommé `cwd` (pour *current working directory*) qui pointe vers le répertoire courant du processus. Il suffit alors de lire le contenu de ce lien symbolique (que nous étudierons dans le prochain chapitre) pour connaître le chemin recherché. Cette information n'est toutefois disponible que pour le propriétaire du processus concerné.

Pour simplifier encore le travail, il existe un sous-répertoire `/proc` nommé `self`, qui correspond au processus appelant. Voici donc un moyen simple d'accéder au répertoire courant :

```
$ cd /etc
$ ls -l /proc/self/
total 0
-r--r--r--   1 ccb      ccb     0 Nov 29 18:46 cmdline
lrwx------   1 ccb      ccb     0 Nov 29 18:46 cwd -> /etc
-r--------   1 ccb      ccb     0 Nov 29 18:46 environ
lrwx------   1 ccb      ccb     0 Nov 29 18:46 exe -> /bin/ls
   [...]
$ cd /usr/local/sbin/
$ ls -l /proc/self/
```

```
total 0
-r--r--r--   1 ccb      ccb     0 Nov 29 18:46 cmdline
lrwx------   1 ccb      ccb     0 Nov 29 18:46 cwd -> /usr/local/sbin
-r--------   1 ccb      ccb     0 Nov 29 18:46 environ
lrwx------   1 ccb      ccb     0 Nov 29 18:46 exe -> /bin/ls
   [...]
$
```

Il faut remarquer que lorsque le pseudo-système de fichiers `/proc` n'est pas accessible, la bibliothèque C doit se rabattre sur la méthode usuelle en remontant de répertoire en répertoire jusqu'à la racine.

Les fonctions de lecture du chemin courant renvoient un pointeur sur le buffer contenant le résultat, ou `NULL` en cas d'échec. Dans ces cas-là, la variable globale `errno` renferme le type d'erreur. Généralement, il s'agit de `EINVAL` si on a transmis un pointeur illégal, `ERANGE` si le buffer est trop petit, mais on peut aussi rencontrer `EACCES`. Ce dernier cas est assez rare ; c'est une situation où on se trouve dans un répertoire sur lequel on a le droit d'exécution (donc de parcours) mais pas de lecture, et où le pseudo-système de fichiers `/proc` n'est pas monté. Normalement, ceci ne devrait pas se produire sur les systèmes Linux actuels.

Les applications courantes ont rarement besoin de changer de répertoire de travail. Les boîtes de dialogue graphiques pour le chargement ou la sauvegarde de données travaillent en effet avec des chemins d'accès absolus (depuis la racine) ou relatifs (depuis le répertoire courant), mais ne nécessitent pas de changement de répertoire.

Il existe toutefois des processus qui fonctionnent, comme les démons, pendant de longue période, en arrière-plan, en se faisant oublier de l'utilisateur. Il faut absolument qu'une telle application revienne à la racine du système de fichiers lors de son initialisation. En effet, dans le cas contraire, il serait impossible de démonter le système de fichiers sur lequel elle se trouve. Par exemple, un démon lancé par un utilisateur depuis son répertoire `/home/abc` ne doit en aucun cas empêcher l'administrateur de démonter temporairement la partition `/home` si le besoin se fait sentir. Celui-ci serait obligé d'avoir recours à l'utilitaire `fuser -k` pour tuer le processus bloquant le système de fichiers. Dans ce type de logiciel, on introduira donc un `chdir("/")` en début de programme.

Le programme d'exemple ci-dessous va servir à montrer le comportement de `fchdir()`, qui est légèrement plus compliqué que celui de `chdir()` puisqu'il faut passer par l'ouverture du répertoire avec `open()`. Sous Linux, il existe d'ailleurs un attribut `O_DIRECTORY` pour `open()`, servant à faire échouer cet appel système s'il est invoqué sur autre chose qu'un répertoire. Nous n'avons pas employé cet argument car il n'est pas portable, et les développeurs du noyau précisent qu'il ne doit être utilisé que pour l'implémentation de la fonction de bibliothèque `opendir()`.

Un deuxième point intéressant avec cet exemple est la manière de traiter l'erreur `ERANGE` de `getcwd()` pour augmenter la taille du buffer fourni. Nous utilisons délibérément une taille ridiculement petite au début (16 caractères) pour obliger la routine à réallouer automatiquement une nouvelle zone mémoire.

exemple_fchdir.c :

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
```

```
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>

    void
affiche_chemin_courant (void)
{
    char * chemin = NULL;
    char * nouveau = NULL;
    int    taille = 16;

    while (1) {
        if ((nouveau = realloc(chemin, taille)) == NULL) {
            perror("realloc");
            break;
        }
        chemin = nouveau;
        if (getcwd(chemin, taille) != NULL) {
            fprintf(stdout, "%s\n", chemin);
            break;
        }
        if (errno != ERANGE) {
            perror("getcwd");
            break;
        }
        taille *= 2;
    }
    if (chemin != NULL)
        free(chemin);
}

    void
change_chemin_courant (const char * nom)
{
    int  fd;
    if ((fd = open(nom, O_RDONLY)) < 0) {
        perror(nom);
        return;
    }
    if (fchdir(fd) < 0)
        perror(nom);
    close(fd);
}

    int
main (int argc, char * argv[])
{
    int i;

    affiche_chemin_courant();
    for (i = 1; i < argc; i++) {
        change_chemin_courant(argv[i]);
```

```
        affiche_chemin_courant();
    }
    return EXIT_SUCCESS;
}
```

Lors de l'exécution, nous allons nous déplacer dans plusieurs répertoires, dont les noms mesurent plus de 16 caractères.

```
$ cd /usr/local/bin
$ ./exemple_fchdir /etc /usr/X11R6/include/X11/bitmaps/ /etc/inittab
/usr/local/bin
/etc
/usr/X11R6/include/X11/bitmaps
/etc/inittab: N'est pas un répertoire
/usr/X11R6/include/X11/bitmaps
$ pwd
/usr/local/bin
$
```

La tentative de déplacement vers `/etc/inittab` (qui est un fichier et pas un répertoire) échoue évidemment. Nous voyons aussi que les changements de répertoire courant du processus exécutable n'ont bien entendu pas affecté le répertoire de travail du shell, comme le montre la commande `pwd` invoquée finalement.

Lorsqu'un programme reçoit un nom de fichier (quelle que soit la méthode utilisée), il arrive qu'il ait besoin de connaître son emplacement précis sur le système. Le chemin transmis peut en effet contenir des références relatives au répertoire courant (`../../src/`) ou utiliser des liens symboliques entre différents répertoires. Pour « nettoyer » un chemin d'accès, il existe une fonction **`realpath()`** issue de l'univers BSD, mais définie par SUSv3. Suivant les systèmes Unix et les versions de la bibliothèque C, elle peut être déclarée dans `<unistd.h>` ou dans `<stdlib.h>`. Les dernières versions de la GlibC emploient ce dernier fichier d'en-tête.

```
char * realpath (char * chemin, char * chemin_exact);
```

Le premier argument est la chaîne contenant le chemin qu'on désire traiter. Le second argument est un tableau comprenant au minimum **`MAXPATHLEN`** caractères, cette constante étant définie dans `<sys/param.h>`. Ce tableau sera rempli par la fonction `realpath()`, qui renverra un pointeur sur lui si elle réussit, ou `NULL` en cas d'erreur.

exemple_realpath.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/param.h>

    int
main (int argc, char * argv[])
{
    char chemin_complet[MAXPATHLEN];
    int i;

    for (i = 0; i < argc; i ++) {
        fprintf(stderr, "%s : ", argv[i]);
```

```
        if (realpath(argv[i], chemin_complet) == NULL)
            perror("");
        else
            fprintf(stderr, "%s\n", chemin_complet);
    }
    return EXIT_SUCCESS;
}
```

L'exécution suivante montre que `realpath()` peut aussi bien résoudre les références relatives, comme celles qui sont contenues dans le nom du programme exécutable transmis en argument `argv[0]` de `main()`, que les liens symboliques comme `/usr/tmp` qui pointe traditionnellement vers `/var/tmp`.

```
$ ./exemple_realpath /usr/tmp/
./exemple_realpath : /home/ccb/Doc/ProgLinux/Exemples/20/exemple_realpath
/usr/tmp/ : /var/tmp
$
```

Cette fonction peut être très commode dans certains cas, mais elle est toutefois peu conseillée car sa portabilité n'est pas assurée du fait qu'elle n'est pas définie par Posix.

Changement de répertoire racine

Chaque processus dispose d'un pointeur sur son répertoire racine dans le système de fichiers. Pour la plupart des processus, il s'agit du véritable répertoire de départ de toute l'arborescence du système. Il peut toutefois être utile dans certaines conditions de modifier le répertoire qu'un processus considère comme la racine du système de fichiers.

Dans [CHESWICK 1991] *An Evening With Berferd*, Bill Cheswick décrit un piège qu'il a construit pour étudier un pirate. Il établit dans un répertoire banal une fausse arborescence avec les sous-répertoires habituels minimaux, et utilise l'appel système **`chroot()`** pour que son visiteur indésirable croit se trouver dans le véritable système de fichiers complet.

L'appel système `chroot()` est une fonction privilégiée demandant la capacité `CAP_SYS_CHROOT`. Il n'y aurait probablement pas de grand risque à la laisser à la disposition des utilisateurs courants, sauf peut-être pour sa capacité à construire des chevaux de Troie destinés à piéger les mots de passe d'un autre utilisateur (en écrivant un faux `/bin/su`). L'application la plus courante de cet appel système est celle qui est utilisée dans le démon de `ftp` anonyme. Lorsqu'une connexion est établie, le processus bascule sur une nouvelle racine du système de fichiers en `/home/ftp`. Dans ce répertoire, on retrouve les utilitaires indispensables de `/bin` et les bibliothèques partagées de `/lib` (respectivement dans `/home/ftp/bin` et dans `/home/ftp/lib`).

Dans l'exemple suivant, nous allons créer un programme Set-UID *root* qui se déplace dans le répertoire indiqué en premier argument, en fait son répertoire racine, et exécute les commandes passées dans les arguments suivants.

exemple_chroot.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
    int
main (int argc, char * argv[])
{
    if (argc < 3) {
        fprintf(stderr, "Syntaxe %s chemin commande...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (chdir(argv[1]) != 0){
        perror("chdir");
        exit(EXIT_FAILURE);
    }
    if (chroot(argv[1]) != 0) {
        perror("chroot");
        exit(EXIT_FAILURE);
    }
    if (seteuid(getuid()) < 0) {
        perror("seteuid");
        exit(EXIT_FAILURE);
    }
    execvp(argv[2], argv + 2);
    perror("execvp");
    return EXIT_FAILURE;
}
```

Nous allons demander à ce programme, après l'avoir installé Set-UID *root*, de lancer la commande `sh`. Pour que cela fonctionne, il faut qu'il puisse trouver ce fichier exécutable dans `/bin` et les bibliothèques partagées nécessaires dans `/lib`. Le plus simple est de changer notre racine pour aller dans `/home/ftp`. Nous vérifierons alors par un `cd /` que nous sommes bien restés à l'emplacement prévu.

```
$ ls -l /home/ftp
total 4
d--x--x--x   2 root     root         1024 Aug 12 15:40 bin
d--x--x--x   2 root     root         1024 Aug 12 15:40 etc
drwxr-xr-x   2 root     root         1024 Aug 12 15:40 lib
drwxr-sr-x   2 root     ftp          1024 Nov  7 23:01 pub
$ su
Password:
# chown root.root exemple_chroot
# chmod u+s exemple_chroot
# exit
$ ./exemple_chroot /home/ftp/ sh
$ cd /
$ ls -l
total 4
d--x--x--x   2 root     root         1024 Aug 12 15:40 bin
d--x--x--x   2 root     root         1024 Aug 12 15:40 etc
drwxr-xr-x   2 root     root         1024 Aug 12 15:40 lib
drwxr-sr-x   2 root     ftp          1024 Nov  7 23:01 pub
$ exit
$
```

Création et suppression de répertoire

Pour créer un nouveau répertoire, ou en supprimer un, il existe deux appels système, **mkdir()** et **rmdir()**, dont le fonctionnement est assez intuitif et rappelle les deux commandes /bin/mkdir et /bin/rmdir qui sont construites à partir de ces fonctions. Leurs prototypes sont déclarés ainsi dans <unistd.h> :

```
int mkdir (const char * repertoire, mode_t mode);
int rmdir (const char * repertoire);
```

L'emploi du type mode_t pour le second argument de mkdir() nécessite l'inclusion supplémentaire de <fcntl.h> et de <sys/types.h>, comme avec open().

Ces deux appels système renvoient 0 s'ils réussissent, et –1 en cas d'échec. En plus des erreurs liées aux autorisations d'accès ou aux irrégularités concernant le nom fourni, mkdir() peut échouer avec le code ENOSPC dans errno si le disque est saturé ou si le quota attribué à l'utilisateur est rempli, ou avec l'erreur EEXIST si le répertoire existe déjà. De son côté, rmdir() peut renvoyer surtout les erreurs EACCES ou EPERM liées aux autorisations d'accès, ou ENOTEMPTY si le répertoire à supprimer n'est pas vide, ou EBUSY si on essaye de supprimer le répertoire de travail courant d'un autre processus. Cette dernière erreur n'est pas respectée sur tous les systèmes.

La profondeur des sous-répertoires dans une arborescence n'est pas limitée. Il est donc possible de créer des sous-répertoires imbriqués jusqu'à la saturation du disque. Il peut toutefois y avoir des limitations liées au système de fichiers sous-jacent. Par exemple, les systèmes ISO 9660, sans les extensions *Rock Ridge*, ne permettent pas plus de huit niveaux de sous-répertoires. Ce système de fichiers n'est toutefois utilisé que pour les CD-Rom, et il n'y a donc pas de raisons d'invoquer mkdir() dessus[1].

Le mode fourni en second argument de mkdir() sert à indiquer les autorisations d'accès du répertoire nouvellement créé. Comme pour open(), on utilise les constantes S_Ixxx ou leurs valeurs octales que nous avons vues dans le chapitre 19. Avec un répertoire, les différents bits d'autorisation ont les significations suivantes :

Bit	Valeur	Signification
S_ISGID	02000	Bit Set-GID : les fichiers ou les sous-répertoires créés dans ce répertoire appartiendront automatiquement au même groupe que lui.
S_ISVTX	01000	Bit « Sticky » : les fichiers créés dans ce répertoire ne pourront être écrasés ou effacés que par leur propriétaire ou celui du répertoire. C'est utile pour des répertoires comme /tmp ou des zones de stockage publiques comme /pub/incoming d'un serveur ftp anonyme.
S_IR…	0..4..	Lecture : on a accès au contenu du répertoire.
S_IW…	0..2..	Écriture : on peut créer un fichier ou un sous-répertoire dans le répertoire.
S_IX…	0..1..	Exécution : on peut entrer dans le répertoire pour en faire son répertoire de travail.

1. L'application mkisofs est plutôt mal nommée, car elle permet uniquement de créer une image d'un répertoire au format Iso 9660, mais ne crée pas un système de fichiers dans lequel on pourrait écrire.

On trouvera donc le plus souvent les valeurs suivantes :

- 00755 : répertoire normal, lisible, accessible en déplacement pour tous, écriture uniquement par le propriétaire ;
- 00700 : répertoire privé, accessible uniquement par son propriétaire (parfois `/root`) ;
- 01777 : répertoire `/tmp` par exemple.

Bien entendu, le fait d'interdire le parcours ou à plus forte raison la lecture d'un répertoire empêche également l'accès à tous ses sous-répertoires.

Lors de la création d'un nouveau répertoire, les autorisations fournies sont passées au travers du *umask* du processus. Plus précisément, la valeur du *umask* est extraite des permissions demandées. Si le *umask* vaut 0002 (ce qui est courant) et qu'on demande une création 00777, le répertoire aura en réalité la permission 0775. Il faut donc faire attention à modifier son propre *umask* (nous le détaillerons dans le prochain chapitre) si on essaye de créer des répertoires accessibles à tous.

L'exemple suivant met en relief ce comportement. Nous essayons à deux reprises de créer un répertoire en mode 00777 et nous vérifions le résultat en invoquant `ls`. La première tentative se fait sans modifier le *umask*, la seconde après l'avoir ramené à zéro.

exemple_mkdir.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>

    int
main (void)
{
    fprintf(stderr, "Création repertoire mode rwxrwxrwx : ");
    if (mkdir("repertoire", 0777) != 0) {
        perror("");
        exit(EXIT_FAILURE);
    } else {
        fprintf(stderr, "Ok\n");
    }
    system("ls -ld repertoire");
    fprintf(stderr, "Suppression repertoire : ");
    if (rmdir("repertoire") != 0) {
        perror("");
        exit(EXIT_FAILURE);
    } else {
        fprintf(stderr, "Ok\n");
    }
    fprintf(stderr, "Modification umask\n");
    umask(0);
    fprintf(stderr, "Création repertoire mode rwxrwxrwx : ");
    if (mkdir("repertoire", 0777) != 0) {
        perror("");
        exit(EXIT_FAILURE);
```

```
    } else {
        fprintf(stderr, "Ok\n");
    }
    system("ls -ld repertoire");
    fprintf(stderr, "Suppression repertoire : ");
    if (rmdir("repertoire") != 0) {
        perror("");
        exit(EXIT_FAILURE);
    } else {
        fprintf(stderr, "Ok\n");
    }
    return EXIT_SUCCESS;
}
```

Voici l'exécution de ce programme, montrant bien l'influence du *umask* :

```
$ ./exemple_mkdir
Création repertoire mode rwxrwxrwx : Ok
drwxrwxr-x   2 ccb  ccb      1024 Nov 30 14:57 repertoire
Suppression repertoire : Ok
Modification umask
Création repertoire mode rwxrwxrwx : Ok
drwxrwxrwx   2 ccb  ccb      1024 Nov 30 14:57 repertoire
Suppression repertoire : Ok
$
```

Suppression ou déplacement de fichiers

Pour bien comprendre le comportement des fonctions de suppression ou de déplacement de fichiers, il est nécessaire d'observer la structure des données sur un système de fichiers Unix. Sur le disque, les répertoires sont en réalité de simples listes de noms de fichiers, auxquels sont associés des numéros d'i-nœuds. Un i-nœud est un identifiant unique pour un fichier sur le disque. Par contre, un même fichier peut avoir plusieurs noms. Il existe une table globale des i-nœuds, permettant de retrouver le contenu réel du fichier.

Dans l'exemple suivant, nous allons créer deux fichiers – en copiant des fichiers système accessibles à tous –, puis utiliser l'utilitaire `link` pour ajouter plusieurs autres liens physiques sur le même fichier. L'option –`i` de la version Gnu de `ls` nous permettra d'observer les numéros d'i-nœuds associés aux entrées du répertoire. Nous vérifierons donc que le même fichier physique dispose de plusieurs noms indépendants.

```
$ cp /etc/services ./un
$ cp /etc/host.conf ./deux
$ ls -l
total 13
-rw-r--r--   1 ccb      ccb        26 Nov 30 19:04 deux
-rw-r--r--   1 ccb      ccb     11279 Nov 30 19:04 un
$ ln un lien_sur_un
$ ln deux lien_sur_deux
$ ln deux autre_lien_sur_deux
$ ls -l
total 27
```

```
-rw-r--r--   3 ccb      ccb        26 Nov 30 19:04 autre_lien_sur_deux
-rw-r--r--   3 ccb      ccb        26 Nov 30 19:04 deux
-rw-r--r--   3 ccb      ccb        26 Nov 30 19:04 lien_sur_deux
-rw-r--r--   2 ccb      ccb     11279 Nov 30 19:04 lien_sur_un
-rw-r--r--   2 ccb      ccb     11279 Nov 30 19:04 un
$ ls -i
 649300 autre_lien_sur_deux
 649300 lien_sur_deux
 649298 un
 649300 deux
 649298 lien_sur_un
$ rm un
$ rm lien_sur_deux
$ ls -l
total 14
-rw-r--r--   2 ccb      ccb        26 Nov 30 19:04 autre_lien_sur_deux
-rw-r--r--   2 ccb      ccb        26 Nov 30 19:04 deux
-rw-r--r--   1 ccb      ccb     11279 Nov 30 19:04 lien_sur_un
$
```

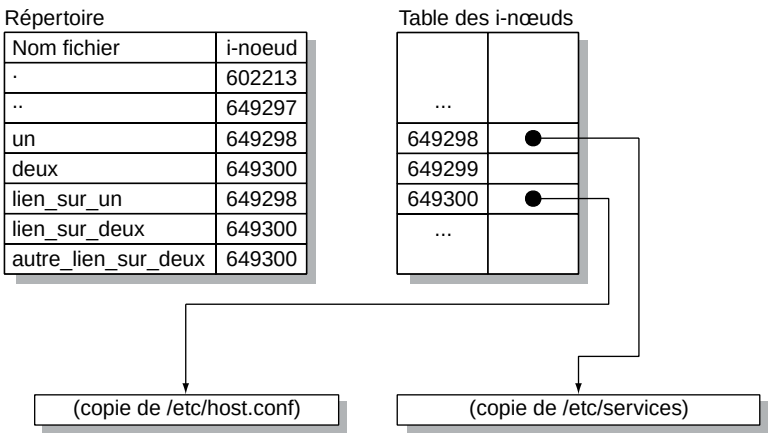


Figure 20.1
Noms dans un répertoire et fichiers physiques

Dans le i-nœud correspondant à un fichier est mémorisé le nombre d'entrées de répertoires faisant référence à ce fichier, c'est-à-dire le nombre de noms différents dont un fichier dispose. Ce nombre est affiché dans la deuxième colonne de la commande `ls -l`. Lorsque le nombre de liens tombe à zéro, le fichier est effectivement effacé du disque s'il n'est ouvert par aucun processus, mais il ne l'est pas avant.

L'appel système permettant d'effacer un fichier est donc nommé **`unlink()`**, et non `erase()`, `delete()` ou quelque chose dans ce goût-là, car il sert uniquement à supprimer le lien entre un nom de fichier (une entrée de répertoire) et l'i-nœud correspondant au contenu du fichier. Cet appel est déclaré dans `<unistd.h>` ainsi :

```
int unlink(const char * nom_fichier) ;
```

Dans l'exemple suivant, nous allons créer un fichier, puis le supprimer tout en le conservant ouvert. Nous contrôlerons que son nom disparaît du répertoire (en invoquant `ls -l`), mais que nous pouvons continuer à accéder à son contenu.

exemple_unlink.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    FILE * fp;
    char   chaine[27];

    fprintf(stdout, "Création fichier\n");
    fp = fopen("essai.unlink", "w+");
    if (fp == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fprintf(fp, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    fflush(fp);
    system("ls -l essai.unlink");
    fprintf(stdout, "Effacement fichier\n");
    if (unlink("essai.unlink") < 0) {
        perror("unlink");
        exit(EXIT_FAILURE);
    }
    system("ls -l essai.unlink");
    fprintf(stdout, "Relecture du contenu du fichier\n");
    if (fseek(fp, 0, SEEK_SET) < 0) {
        perror("fseek");
        exit(EXIT_FAILURE);
    }
    if (fgets(chaine, 27, fp) == NULL) {
        perror("fgets");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Lu : %s\n", chaine);
    fprintf(stdout, "Fermeture fichier\n");
    fclose(fp);
    return EXIT_SUCCESS;
}
```

L'exécution confirme nos attentes, le fichier a bien disparu du répertoire lors du second appel de `ls`, mais on peut continuer à en lire le contenu tant qu'on ne l'a pas refermé.

```
$ ./exemple_unlink
Création fichier
-rw-rw-r--   1 ccb      ccb        26 Dec  1 14:24 essai.unlink
Effacement fichier
```

```
ls: essai.unlink: Aucun fichier ou répertoire de ce type
Relecture du contenu du fichier
Lu : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Fermeture fichier
$
```

L'appel système `rmdir()` permet de supprimer des répertoires, et l'appel `unlink()` des fichiers. Il existe une fonction de la bibliothèque C nommée **`remove()`** qui leur sert de frontal en invoquant l'appel système correspondant au type d'objet concerné. Elle est déclarée dans `<stdio.h>` :

```
int remove (const char * nom)
```

Son emploi est très simple, le programme suivant efface les fichiers et répertoires dont les noms sont transmis en argument.

exemple_remove.c :

```
    #include <stdio.h>

    int
main (int argc, char * argv[])
{
    char chaine[5];
    int  i;

    for (i = 1; i < argc; i ++) {
        fprintf(stderr, "Effacer %s ? ", argv[i]);
        if (fgets(chaine, 5, stdin) == NULL)
            break;
        if ((chaine[0] != 'o') && (chaine[0] != 'O'))
            continue;
        if (remove(argv[i]) < 0)
            perror(argv[i]);
    }
    return EXIT_SUCCESS;
}
```

L'exemple d'exécution suivant se déroule comme prévu :

```
$ touch essai.remove.fichier
$ mkdir essai.remove.repertoire
$ ls -dF essai.remove.*
essai.remove.fichier  essai.remove.repertoire/
$ ./exemple_remove essai.remove.*
Effacer essai.remove.fichier ? o
Effacer essai.remove.repertoire ? o
$ ls -dF essai.remove.*
ls: essai.remove.*: Aucun fichier ou répertoire de ce type
$
```

Pour déplacer ou renommer un fichier ou un répertoire, il existe un appel système unique, **`rename()`**, facile d'utilisation :

```
int rename (const char * ancien_nom, const char * nouveau_nom);
```

Cette routine renvoie 0 si elle réussit ou –1 en cas d'erreurs, qui, en dehors des problèmes habituels de pointeurs invalides ou de permissions d'accès, peuvent être :

- `EBUSY` : le répertoire qu'on veut écraser ou celui qu'on veut déplacer est utilisé comme répertoire de travail par un processus.
- `EINVAL` : on essaye de déplacer un répertoire vers un de ses propres sous-répertoires. C'est impossible.
- `EISDIR` : on essaye d'écraser un répertoire existant avec un fichier régulier.
- `ENOTEMPTY` : le répertoire qu'on veut écraser n'est pas vide.
- `EXDEV` : on essaye de déplacer un fichier ou un répertoire vers un système de fichiers différent. C'est impossible, il faut passer par une étape de copie, puis de suppression.

Si le fichier ou le répertoire cible existe déjà, il est écrasé. Le noyau s'arrange pour que le déplacement soit atomique et que le nouveau nom ne soit jamais absent du système de fichiers.

Notons que ni `rmdir()`, ni `remove()`, ni `rename()` ne sont capables de supprimer ou d'écraser un répertoire non vide. Pour cela, il est nécessaire de descendre récursivement jusqu'au dernier sous-répertoire, puis de remonter en effaçant tout le contenu de chaque sous-répertoire, avec `remove()` par exemple. C'est ce que fait par exemple la commande `rm -r`.

Fichiers temporaires

Il est très fréquent d'avoir besoin de fichiers temporaires. Ne serait-ce que pour insérer des données au milieu d'un fichier existant, la méthode la plus simple consiste à recopier le fichier original dans un fichier temporaire en ajoutant au passage les nouvelles informations, puis à recopier ou à renommer le fichier temporaire pour écraser l'original.

Il existe plusieurs fonctions pouvant nous aider à obtenir un fichier temporaire, mais elles doivent être utilisées avec précaution. En effet, une application écrite proprement doit éviter à tout prix de laisser des fichiers traîner bien après sa terminaison (dans `/tmp` ou ailleurs). Ce n'est pas toujours simple, surtout si le programme peut être tué abruptement par un signal.

Le premier point consiste à obtenir un nom de fichier unique. Ce nom doit être créé par le système, ce qui nous garantit qu'il ne sera pas réattribué lors d'une autre demande de fichier temporaire. Il existe principalement trois fonctions pouvant remplir ce rôle, **`tempnam()`**, **`tmpnam()`** et **`mktemp()`**, les deux premières étant déclarées dans `<stdio.h>`, la dernière dans `<stdlib.h>` :

```
char * tempnam (const char * repertoire_temporaire, const char * prefixe);
char * tmpnam (char * chaine);
char * mktemp (char * motif);
```

Ces trois fonctions renvoient `NULL` si elles échouent et un pointeur sur la chaîne contenant le nom temporaire sinon. La première fonction, `tempnam()`, s'occupe d'allouer l'espace nécessaire en utilisant `malloc()`. Le pointeur renvoyé devra donc être libéré ultérieurement avec `free()`. La seconde fonction, `tmpnam()`, remplit la chaîne passée en argument, qui doit contenir au moins **`L_tmpnam`** octets. Cette valeur est définie dans `<stdio.h>`. Si on lui passe un pointeur `NULL`, `tmpnam()` stockera le nom temporaire dans une zone de mémoire statique, écrasée à chaque appel. Comme tout ceci est dangereux dans un contexte multithread, il

existe une fonction **tmpnam_r()**, avec la même interface mais qui n'accepte pas d'argument NULL (devenant ainsi *a fortiori* réentrante).

Enfin, la fonction mktemp() modifie le contenu de la chaîne qu'on lui passe en argument. Cette dernière doit contenir le motif XXXXXX en guise des six derniers caractères ; ils seront écrasés par une valeur unique lors de l'appel de la fonction. Il ne faut donc pas passer une chaîne constante en argument de mktemp().

Les comportements de ces trois fonctions pour obtenir un nom unique sont légèrement différents :

- tempnam() utilise les arguments qu'on lui transmet (s'ils ne sont pas nuls) pour composer le nom de fichier temporaire. Cette fonction essaye de créer un fichier dans les répertoires suivants, successivement : le contenu de la variable d'environnement TMPDIR, le premier argument qu'on lui passe, le répertoire correspondant à la constante **P_tmpdir** de <stdio.h>, et finalement /tmp/. Ensuite, tempnam() se sert du préfixe fourni pour composer le nom du fichier.
- tmpnam() donne un nom de fichier temporaire dans le répertoire correspondant à la constante P_tmpdir définie dans <stdio.h> (/tmp/ avec la bibliothèque GlibC).
- mktemp(), nous l'avons indiqué, se borne à remplacer les six derniers X du motif fourni en argument pour créer un nom de fichier.

Voici à présent un exemple d'utilisation de ces trois routines.

exemple_temp.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char * nom_tempnam;
    char   nom_tmpnam[L_tmpnam];
    char   nom_mktemp[20];

    nom_tempnam = tempnam(NULL, "abcdef");
    fprintf(stderr, "tempnam(NULL, \"abcdef\") = ");
    if (nom_tempnam == NULL)
        perror("NULL");
    else
        fprintf(stderr, "%s\n", nom_tempnam);
    free(nom_tempnam);

    fprintf(stderr, "tmpnam() = ");
    if (tmpnam(nom_tmpnam) == NULL)
        perror("NULL");
    else
        fprintf(stderr, "%s\n", nom_tmpnam);

    strcpy(nom_mktemp, "/tmp/abcdefXXXXXX");
    fprintf(stderr, "mktemp(\"/tmp/abcdefXXXXXX\") = ");
```

```
    if (mktemp(nom_mktemp) == NULL)
        perror("NULL");
    else
        fprintf(stderr, "%s\n", nom_mktemp);

    return EXIT_SUCCESS;
}
```

On notera que, contrairement à ce qui est affiché durant l'exécution du programme, on ne passe pas la chaîne (constante) "/tmp/abcdefXXXXXX" à mktemp(), mais un tableau de caractères qu'elle peut modifier.

```
$ ./exemple_temp
tempnam(NULL, "abcdef") = /tmp/abcdetfhapc
tmpnam() = /tmp/filei72uIe
mktemp("/tmp/abcdefXXXXXX") = /tmp/abcdefXzqENg
$
```

Une fois qu'on a obtenu un nom de fichier, encore faut-il l'ouvrir effectivement. Cette opération s'effectue avec open() ou fopen(), comme nous l'avons vu dans les chapitres 18 et 19. Un problème peut toutefois se poser. Le système nous garantit uniquement que les fonctions tempnam(), tmpnam() et mktemp() vont renvoyer un nom n'existant pas dans le système de fichiers. Il existe donc une condition de concurrence risquée si on considère qu'un autre processus peut très bien créer le fichier entre le moment du retour de la fonction fournissant le nom et l'appel de open() ou de fopen() suivant. Même si cette situation a très peu de chances de se produire par hasard, elle peut toujours être exploitée pour créer un trou de sécurité dans un logiciel.

Il est donc nécessaire de s'assurer que le fichier sera ouvert de manière exclusive, en employant l'attribut O_EXCL de open() ou l'extension Gnu x de fopen(), ou encore la fonction fopen_exclusif() que nous avons écrite dans le chapitre précédent. On exécute donc quelque chose comme :

```
    char * nom;
    int    fd;

    while (1) {
        nom = tempnam(repertoire_temporaire, prefixe_application);
        if (nom == NULL) {
            perror("tempnam");
            exit(EXIT_FAILURE);
        }
        fd = open(nom, O_CREAT | O_EXCL | O_RDWR, 0600);
        free(nom);
        if (fd >= 0)
            break ;
    }
    /* utilisation du descripteur fd */
```

Toutefois, on peut également utiliser la fonction **mkstemp()**, qui est définie dans <stdlib.h> :

```
int mkstemp (char * motif) ;
```

Comme `mktemp()`, cette fonction modifie le motif fourni, en remplaçant les six derniers `X` par une chaîne aléatoire. Ensuite, elle ouvre le fichier de manière exclusive, en mode de lecture et écriture, puis renvoie le descripteur obtenu. Bien entendu, si on désire obtenir un flux et non un descripteur, on peut employer la fonction `fdopen()` déjà étudiée.

Quelle que soit la méthode choisie, il est important de bien se rappeler qu'on désire obtenir un fichier temporaire, ce qui signifie qu'il faut impérativement l'effacer lorsque le programme se termine. Il est particulièrement agaçant pour un administrateur système de voir le répertoire `/tmp` contenir une myriade de fichiers difficiles à distinguer les uns des autres et qu'il faut effacer manuellement de temps à autre (même si on peut automatiser la suppression en vérifiant la date du dernier accès au fichier à l'aide de la commande `find`).

Une application soignée doit s'assurer d'effacer tous les fichiers temporaires qu'elle crée. Pour cela, la méthode la plus simple consiste à demander au système d'éliminer le fichier aussitôt après son ouverture. On se souvient en effet que l'appel système `unlink()` ne fait disparaître le contenu d'un fichier qu'une fois qu'il n'est plus ouvert par aucun processus et qu'il n'a plus de nom dans le système de fichiers. Tant que nous conserverons notre descripteur ou notre flux ouvert, le fichier temporaire sera donc utilisable. Par contre, dès sa fermeture ou la fin du programme, le fichier disparaîtra définitivement. Ceci permet de pallier le problème d'une terminaison violente de l'application par un signal.

Nous utiliserons donc un code comme celui-ci :

```
int fd;
char motif[20];

strcpy(motif, "/tmp/XXXXXX");
fd = mkstemp(motif);
if (fd < 0) {
    perror("mkstemp");
    exit(EXIT_FAILURE);
}
unlink(motif);
/* Utilisation de fd */
    ...
```

Le fichier sera donc éliminé automatiquement lors de la fin du programme. Il existe une fonction **`tmpfile()`**, définie par SUSv3, qui réalise le même travail, en renvoyant un flux de données. Elle est déclarée dans `<stdio.h>` :

```
FILE * tmpfile(void);
```

Cette routine gère entièrement la création du nom de fichier, l'ouverture exclusive d'un flux et la suppression automatique du fichier temporaire. Son seul défaut est que le nom du fichier n'est pas accessible. Il n'est pas possible de le fournir en argument lors d'une invocation avec `system()` d'une autre application (comme `sort` pour trier le fichier).

On notera également que `tmpfile()` existe sur tous les environnements compatibles Ansi C, mais qu'à la différence de SUSv3, un programme se terminant anormalement sur ces systèmes ne détruira pas nécessairement ses fichiers temporaires.

Recherche de noms de fichiers

Correspondance simple d'un nom de fichier

Lorsqu'on recherche l'ensemble des fichiers dont les noms correspondent à un motif donné, il est possible d'utiliser les routines de manipulation d'expressions régulières que nous avons vues dans le chapitre 16. Toutefois, comme nous l'avons déjà fait remarquer, la syntaxe des expressions régulières n'est pas celle qui est communément adoptée par les shells pour identifier les fichiers. Pour répondre à ce besoin, la bibliothèque C met à notre disposition la fonction **fnmatch()**, mieux adaptée à la comparaison des noms de fichiers et définie dans `<fnmatch.h>` :

```
int fnmatch(const char * motif, const char * nom, int attributs);
```

Cette fonction compare tout simplement le motif transmis en premier argument avec le nom de fichier fourni en seconde position, et renvoie 0 si les chaînes correspondent, ou `FNM_NOMATCH` sinon. Sur certains systèmes, cette routine peut également renvoyer une valeur non nulle autre que `FNM_NOMATCH` en cas d'erreur. Ce n'est pas le cas avec la GlibC.

Le troisième argument permet de configurer certaines options par un OU binaire :

Attribut	Signification
`FNM_PATHNAME`	Avec cette option, les caractères slash '/' sont traités de manière particulière : ils ne sont jamais mis en correspondance avec des caractères génériques. Ce comportement est généralement celui désiré quand on cherche à mettre en correspondance des noms de fichiers.
`FNM_FILE_NAME`	Il s'agit d'un synonyme de `FNM_PATHNAME` spécifique à Gnu.
`FNM_PERIOD`	Le caractère point '.' est traité spécifiquement s'il se trouve en début de nom. Dans ce cas en effet, il ne peut pas être mis en correspondance avec un motif générique. Ce comportement est également celui attendu habituellement lors du traitement des noms de fichiers.
`FNM_NOESCAPE`	Cette option désactive l'utilisation du caractère backslash '\' pour supprimer la signification particulière d'un caractère (comme `\*` pour indiquer un astérisque).
`FNM_CASEFOLD`	Cet attribut est une extension Gnu permettant d'ignorer la différence entre les majuscules et les minuscules durant la mise en correspondance.
`FNM_LEADING_DIR`	Cette extension Gnu permet d'autoriser la mise en correspondance si le motif est la partie initiale du nom et que le reste de ce nom commence par '/'. Ceci revient à accepter le motif `/tmp` pour `/tmp/abcd` par exemple. Cette méthode n'est toutefois pas la meilleure pour traiter des descentes de sous-répertoires.

Classiquement, sur un système Unix, les options qu'on utilise sont `FNM_PATHNAME` et `FNM_PERIOD` puisqu'elles permettent de comparer les noms de fichiers de la même manière que les interpréteurs de commandes usuels. Dans l'exemple ci-dessous, nous utiliserons la fonction `scandir()` que nous avons déjà étudiée, mais cette fois la sélection des fichiers à afficher sera réalisée en employant `fnmatch()` et non plus `regexec()`.

exemple_fnmatch.c :

```
#include <dirent.h>
#include <fnmatch.h>
#include <stdio.h>
#include <stdlib.h>

static char * motif = NULL;
```

```
    int
fn_selection (const struct dirent * entree)
{
    if (fnmatch(motif, entree->d_name, FNM_PATHNAME | FNM_PERIOD) == 0)
        return 1;
    return 0;
}

    int
main (int argc, char * argv[])
{
    struct dirent ** liste;
    int   nb_entrees;
    int   i;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s répertoire motif\n", argv[0]);
        exit (EXIT_FAILURE);
    }
    motif = argv[2];
    nb_entrees = scandir(argv[1], & liste, fn_selection, alphasort);
    if (nb_entrees <= 0)
        exit(EXIT_SUCCESS);
    for (i = 0; i < nb_entrees; i ++) {
        fprintf(stdout, "  %s\n", liste[i]->d_name);
        free(liste[i]);
    }
    fprintf(stdout, "\n");
    free(liste);
    return EXIT_SUCCESS;
}
```

Nous vérifions la comparaison sur les points spécifiques aux mises en correspondance des noms de fichiers :

```
$ ./exemple_fnmatch /dev "tty[0-9]*"
  tty0
  tty1
  tty10
  tty11
  tty12
  tty2
  tty3
  tty4
  tty5
  tty6
  tty7
  tty8
  tty9
$ ./exemple_fnmatch /etc/skel/ ".*"
  .
  ..
  .Xdefaults
  .bash_logout
```

```
  .bash_profile
  .bashrc
  .kde
  .kderc
  .screenrc
$
```

Nous voyons bien que le comportement est celui qui est attendu. Bien entendu, les caractères spéciaux comme « * » ou « . » doivent être protégés du shell à l'aide des guillemets pour arriver intacts au cœur de notre programme.

Recherche sur un répertoire total

L'utilisation conjointe de `scandir()` et de `fnmatch()` nous a permis d'extraire une liste de noms de fichiers appartenant à un répertoire donné. Pour accomplir cette tâche automatiquement, la bibliothèque C met à notre disposition les fonctions **`glob()`** et **`globfree()`** qui sont également bien plus puissantes. Elles sont déclarées dans `<glob.h>` ainsi :

```
int glob(const char * motif, int attribut,
         int (* fn_erreur) (const char * chemin, int erreur),
         glob_t * vecteur);
void globfree (glob_t * vecteur);
```

La fonction `glob()` prend successivement les arguments suivants :

- le motif qu'on désire mettre en correspondance ;
- des attributs regroupés par un OU binaire, que nous détaillerons plus bas ;
- une éventuelle fonction d'erreur qui sera invoquée en cas de problème ;
- une structure de type `glob_t` dans laquelle le résultat sera stocké.

Cette fonction recherche tous les fichiers correspondant au motif transmis, depuis le répertoire de travail courant. Bien entendu, si le motif commence par des références relatives (`../home/…`, `bin/…`) ou absolues (`/var/tmp/…`), le répertoire de recherche est modifié en conséquence. L'ensemble des fichiers sélectionnés est stocké dans une table contenue dans la structure `glob_t` fournie en dernier argument. Cette structure contient les membres suivants :

Nom	Type[1]	Signification
`gl_pathc`	`int`	Ce membre contient le nombre de noms ayant été mis en correspondance.
`gl_pathv`	`char **`	Ce champ représente un pointeur sur une table de noms de fichiers ayant été sélectionnés.
`gl_offs`	`int`	Ce champ est rempli avant d'appeler `glob()`. Il contient le nombre d'emplacements libres que la fonction doit laisser au début de la table `gl_pathv`. Il n'est utilisé que si la constante `GLOB_DOOFS` est présente dans les attributs de `glob()`. Sinon, il est ignoré, même s'il n'est pas nul.
`gl_opendir`	`fonction`	Ce membre est une extension Gnu. Il s'agit d'un pointeur sur une fonction permettant de remplacer `opendir()`. Le prototype de cette fonction doit être compatible avec celui de `opendir()`. Ceci est principalement utile pour insérer des routines d'encadrement de débogage.

1. Les types exacts des pointeurs de fonction ne sont pas développés. On se reportera au besoin au prototype de la fonction remplacée pour implémenter une routine ayant la même interface.

Nom	Type[1]	Signification
gl_closedir	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant closedir().
gl_readdir	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant readdir().
gl_stat	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant stat(), que nous étudierons dans le prochain chapitre.
gl_lstat	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant lstat(), que nous étudierons dans le prochain chapitre.

1. Les types exacts des pointeurs de fonction ne sont pas développés. On se reportera au besoin au prototype de la fonction remplacée pour implémenter une routine ayant la même interface.

Il est conseillé de se limiter à l'emploi des trois premiers membres uniquement puisqu'ils sont définis par SUSv3.

Les attributs qu'on peut détailler pour paramétrer le fonctionnement de glob() sont les suivants :

Nom	Signification
GLOB_APPEND	Le résultat doit être ajouté à celui qui a déjà été obtenu dans la structure glob_t par un appel antérieur à glob(). Ceci permet de combiner le résultat de plusieurs recherches (équivalent ainsi à un OU logique). Cet attribut ne doit pas être utilisé lors de la première invocation de glob(). Le pointeur gl_pathv peut être modifié par realloc(), et l'ancien pointeur n'a peut-être plus de signification lors du retour de glob(). Il faut donc bien relire le contenu de ce membre, sans le sauvegarder entre deux appels.
GLOB_ALTDIRFUNC	Cet attribut est une extension Gnu qui indique que glob() doit utiliser les pointeurs de fonctions des membres gl_opendir, gl_readdir, etc., de la structure glob_t. Ceci n'a pas d'utilité dans les applications courantes, mais peut servir à gérer de manière uniforme des répertoires normaux et des pseudo-systèmes de fichiers comme une liaison ftp ou le contenu d'une archive tar.
GLOB_BRACE	Cette extension Gnu demande que les accolades soient employées à la manière du shell csh, c'est-à-dire qu'elles indiquent une liste des différentes possibilités, séparées par des virgules.
GLOB_DOOFS	Lorsque cet attribut est signalé, la valeur du membre gl_offs de la structure glob_t est utilisée pour réserver des emplacements au début de la table gl_pathv. Les pointeurs ainsi réservés sont initialisés à NULL. Si on se sert de cet attribut, il faut le mentionner à chaque invocation successive éventuelle lors d'un GLOB_APPEND. Ceci est utile pour glisser ensuite dans les emplacements libres des chaînes représentant le nom d'un fichier exécutable à invoquer et ses éventuelles options, avant d'appeler execvp() avec le tableau gl_pathv. Ainsi, on peut simuler le développement des caractères génériques du shell avant de lancer un programme.
GLOB_ERR	Quand glob() rencontre une difficulté lors de la lecture d'un répertoire, il abandonne immédiatement si cet attribut est présent. Sinon, il tente de continuer quand même. Nous verrons plus bas qu'on peut indiquer un pointeur sur un gestionnaire d'erreur dans l'invocation de glob() afin d'affiner la détection de problèmes.
GLOB_MARK	Lorsqu'un sous-répertoire correspond au motif transmis, on le stocke en ajoutant un slash à la fin de son nom.

Nom	Signification
GLOB_NOCHECK	Si aucune correspondance n'a pu être établie, renvoyer le motif original en guise de résultat plutôt que d'indiquer un échec.
GLOB_NOESCAPE	Cette option est équivalente à FNM_NOESCAPE de fnmatch() que glob() invoque de manière interne. Elle sert donc à désactiver le comportement particulier du backslash '\', qui permet autrement de protéger les caractères spéciaux.
GLOB_NOMAGIC	Cette extension Gnu permet de renvoyer le motif original si aucune correspondance n'est trouvée, à la manière de GLOB_NOCHECK, mais uniquement si le motif ne contient pas de caractères spéciaux. Dans ce cas, on peut par exemple décider de créer le fichier, chose qui est plus compliquée avec GLOB_NOCHECK seul.
GLOB_NOSORT	Ne pas trier les chemins d'accès par ordre alphabétique. Ceci permet théoriquement de gagner du temps mais, en pratique, le tri en mémoire des noms de fichiers consomme avec les processeurs modernes une durée infime par rapport à la consultation du contenu du répertoire sur le disque.
GLOB_PERIOD	Cette extension Gnu est équivalente à FNM_PERIOD de fnmatch(). Le caractère point '.' en début de nom ne peut pas être mis en correspondance avec un caractère générique.
GLOB_TILDE	Avec cette extension Gnu, le caractère tilde '~' est traité spécialement lorsqu'il apparaît en tête de motif. Comme avec le shell, le tilde seul ou suivi d'un slash correspond au répertoire personnel de l'utilisateur. Si le tilde est suivi d'un nom d'utilisateur, il représente alors son répertoire personnel. Par exemple les chaînes ~/.fvwmrc ou ~ftp/pub/ sont traitées comme le fait le shell. Si le répertoire personnel n'est pas accessible quelle qu'en soit la raison, le tilde est alors considéré comme un caractère normal appartenant au nom du fichier.
GLOB_TILDE_CHECK	Dans cette extension Gnu, le comportement est le même que GLOB_TILDE, à la différence que glob() échoue si la mise en correspondance du tilde avec un répertoire personnel n'est pas possible plutôt que de considérer le tilde comme un caractère normal.

Si une erreur se produit alors que glob() tente de lire le contenu d'un répertoire et si le pointeur de fonction fourni en troisième argument n'est pas NULL, celle-ci sera invoquée avec en arguments le nom du chemin d'accès dont la lecture a échoué et le contenu de la variable globale errno telle qu'elle a été remplie par les fonctions opendir(), readdir(), stat() ou lstat(). Si la fonction d'erreur renvoie une valeur non nulle, ou si l'attribut GLOB_ERR a été indiqué, la fonction glob() se terminera immédiatement. Sinon, elle tentera de passer à la mise en correspondance suivante.

La valeur de retour de glob() est nulle si tout s'est bien passé, ou la fonction renvoie l'une des constantes suivantes en cas d'échec :

- GLOB_ABORTED : la routine a été arrêtée à la suite d'une erreur.
- GLOB_NOMATCH : aucune correspondance n'a pu être établie.
- GLOB_NOSPACE : un manque de mémoire a empêché l'allocation de l'espace nécessaire.

Finalement, les données allouées avec glob() au sein de la structure glob_t peuvent être libérées à l'aide de la fonction globfree().

Le programme suivant va simplement afficher les mises en correspondance avec les chaînes passées en argument.

exemple_glob.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <glob.h>

    int
main (int argc, char * argv[])
{
    glob_t  chemins;
    int     i;
    int     erreur;

    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s motif...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    erreur = glob(argv[1], 0, NULL, & chemins);
    if ((erreur != 0) && (erreur != GLOB_NOMATCH))
        perror(argv[1]);
    for (i = 2; i < argc; i ++) {
        erreur = glob(argv[i], GLOB_APPEND, NULL, & chemins);
        if ((erreur != 0) && (erreur != GLOB_NOMATCH))
            perror(argv[i]);
    }
    for (i = 0; i < chemins . gl_pathc; i ++)
        fprintf(stdout, "%s\n", chemins.gl_pathv[i]);
    globfree(& chemins);
    return EXIT_SUCCESS;
}
```

L'exécution confirme le fonctionnement de `glob()` dans la vérification de répertoires.

```
$ ./exemple_glob "/dev/tty1*" "*lob*"
/dev/tty1
/dev/tty10
/dev/tty11
/dev/tty12
exemple_glob
exemple_glob.c
$
```

Développement complet à la manière d'un shell

Le shell offre bien d'autres fonctionnalités que le simple remplacement des caractères génériques. La bibliothèque C propose un couple de fonctions, **`wordexp()`** et **`wordfree()`**, particulièrement puissantes, qui assurent l'essentiel des tâches accomplies habituellement par le shell. Ces fonctions travaillent sur un modèle assez semblable à celui de `glob()` et de `globfree()`, mais en utilisant une structure de données de type **`wordexp_t`**. Elles sont d'ailleurs déclarées dans `<wordexp.h>`. Le concept ici est en effet de remplacer des mots par leur signification après les interprétations suivantes :

Développement du tilde

En début de chaîne, le caractère '~' seul ou suivi d'un slash représente le répertoire personnel de l'utilisateur appelant, déterminé grâce à la variable d'environnement `HOME`. Si le tilde est directement suivi d'un nom d'utilisateur, déterminé avec la fonction `getpwname()` que nous étudierons dans le chapitre 26, il s'agit du répertoire personnel de celui-ci. Lorsque le tilde apparaît au cœur d'un nom, il est considéré comme un caractère normal.

Substitution des variables

Les chaînes commençant par un `$` sont remplacées par la variable d'environnement correspondante, avec plusieurs syntaxes possibles :

Syntaxe	Substitution
`$VARIABLE`	La valeur de la variable est renvoyée. Le nom de la variable est délimité par le premier caractère blanc rencontré après le `$`
`${VARIABLE}`	La valeur de la variable est directement renvoyée. Les accolades permettent de délimiter le nom, pour pouvoir le joindre à d'autres éléments sans insérer d'espace. Ainsi, si `VAR` vaut `TERNIT`, `E${VAR}E` correspond à `ETERNITE`.
`${#VARIABLE)`	Renvoie le nombre de lettres contenues dans la variable. Ainsi, si la variable `VAR` contient le mot `ETERNITE`, `${#VAR}` renvoie 8.
`${VARIABLE:-DEFAUT}`	Si la variable n'est pas définie ou si elle est vide, renvoyer la valeur par défaut. Sinon renvoyer la valeur de la variable.
`${VARIABLE:=DEFAUT}`	Si la variable n'est pas définie ou si elle est vide, la remplir avec la valeur par défaut. Renvoyer la valeur de la variable.
`${VARIABLE:?MESSAGE}`	Si la variable n'est pas définie ou si elle est vide, afficher le message sur `stderr` et échouer. Sinon renvoyer sa valeur.
`${VARIABLE:+VALEUR}`	Renvoyer la valeur indiquée si la variable est définie et non-vide. Sinon ne rien substituer.
`${VARIABLE##PREFIXE}`	Renvoyer la valeur de la variable en ayant retiré le maximum de caractères correspondant au préfixe fourni. On essaye de supprimer le plus grand nombre de caractères possibles. Ainsi si `VAR` vaut `ETERNITE`, `${VAR##*T}` renvoie `E`, car on supprime tous les caractères jusqu'au second `T`
`${VARIABLE#PREFIXE}`	Comme pour le cas précédent, on supprime le préfixe indiqué, mais en retirant le minimum de lettres. Si `VAR` vaut `ETERNITE`, `${VAR#*T}` renvoie `ERNITE`.
`${VARIABLE%%SUFFIXE}`	Cette fois-ci, on supprime le suffixe indiqué en essayant de retirer le maximum de caractères. `${VAR%%T*}` renvoie uniquement `E` car on retire tout à partir du premier `T`.
`${VARIABLE%SUFFIXE}`	Symétriquement, on retire le plus petit suffixe possible. `${VAR%T*}` renvoie `ETERNI`.

Évaluation arithmétique et exécution de commande

Les chaînes du type `` `*commande*` `` ou `$(*commande*)` sont remplacées par le résultat de la commande qui est exécutée dans un shell, comme avec la commande `system()`.

Les chaînes du type `$[*calcul*]` ou `$((*calcul*))` sont remplacées par le résultat du calcul. On trouvera le détail des opérations arithmétiques possibles dans la page de manuel du shell. Les expressions sont évaluées de gauche à droite.

> **Attention**
> La première syntaxe `$[calcul]` est considérée comme obsolète et ne doit dorénavant plus être employée.

Découpage des mots et développement des noms de fichiers

Finalement, la chaîne est découpée en mots en employant les séparateurs du shell, puis les noms de fichiers sont développés en remplaçant tout mot contenant des caractères génériques par la liste des fichiers dont les noms lui correspondent.

Les fonctions que la bibliothèque C nous fournit pour analyser une chaîne à la manière du shell sont les suivantes :

```
int wordexp (const char * chaine, wordexp_t * mots, int attributs);
void wordfree (wordexp_t * mots);
```

La fonction `wordexp()` prend la chaîne qu'on lui fournit en premier argument, effectue toutes les transformations que nous avons aperçues ci-dessus, et renvoie la liste des mots trouvés dans la structure `wordexp_t` sur laquelle on passe un pointeur en second argument. Cette structure contient les membres suivants :

Nom	Type	Signification
`we_wordc`	`int`	Le nombre de mots contenus dans le tableau suivant.
`we_wordv`	`char **`	Le tableau proprement dit contenant des pointeurs sur des chaînes de caractères correspondant aux différents mots.
`we_offs`	`int`	Comme le champ `gl_offs` de la structure `glob_t`, ce membre permet de réserver de l'espace au début de la table `we_wordv`, à condition d'utiliser l'attribut `WRDE_DOOFFS`.

Les différents attributs qu'on peut transmettre à `wordexp()` sont combinés avec un OU binaire parmi les constantes suivantes :

Nom	Rôle
`WRDE_APPEND`	Ajouter les mots trouvés à ceux qui sont déjà présents dans la structure `wordexp_t`. Cette option ne doit pas être utilisée lors du premier appel de `wordexp()`.
`WRDE_DOOFFS`	Réserver dans la table `we_wordv` la place indiquée dans le membre `we_offs` de la structure `wordexp_t`.
`WRDE_NOCMD`	Ne pas effectuer la substitution de commandes. Ceci évite qu'un programme Set-UID exécute des commandes arbitraires fournies par l'utilisateur. Si on essaye de transmettre une chaîne contenant `` `commande` `` ou `$(commande)`, `wordexp()` échoue.
`WRDE_REUSE`	Réutiliser une structure `wordexp_t` ayant déjà servi. Ceci évite de libérer les données à chaque fois.
`WRDE_SHOWERR`	Lors de la substitution de commandes, sert à utiliser le même flux d'erreur standard que le processus en cours. Ceci permet d'afficher des éventuels messages de diagnostic. Par défaut, ces erreurs ne sont pas visibles.
`WRDE_UNDEF`	Considérer qu'il y a une erreur si on essaye de consulter une variable d'environnement non définie.

La fonction `wordexp()` renvoie zéro si elle réussit, ou l'une des constantes suivantes en cas d'erreur :

- `WRDE_BADCHAR` : la chaîne contient un caractère interdit, comme `<`, `>`, `&`, `;`, `|` ou `\n`.
- `WRDE_BADVAL` : une variable est indéfinie et on a utilisé l'option `WRDE_UNDEF`.
- `WRDE_CMDSUB` : on a essayé de faire une substitution de commandes alors que l'option `WRDE_NOCMD` a été demandée.
- `WRDE_NOSPACE` : pas assez de mémoire pour allouer la table.
- `WRDE_SYNTAX` : une erreur de syntaxe a été détectée, comme des accolades manquantes par exemple.

La fonction `wordfree()` permet bien sûr de libérer la mémoire occupée par les tables contenues dans la structure passée en argument. Dans l'exemple suivant, nous allons construire un microshell n'ayant qu'une seule commande interne, `set`, permettant de configurer une variable d'environnement. Toutes les autres commandes seront exécutées en employant `execvp()`.

exemple_wordexp.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <wordexp.h>
    #include <sys/wait.h>

    void
affiche_erreur (int numero)
{
    switch (numero) {
    case WRDE_BADCHAR :
        fprintf(stderr, "Caractère interdit \n"); break;
    case WRDE_BADVAL :
        fprintf(stderr, "Variable indéfinie \n"); break;
    case WRDE_CMDSUB :
        fprintf(stderr, "Invocation de commande interdite \n"); break;
    case WRDE_NOSPACE :
        fprintf(stderr, "Pas assez de mémoire \n"); break;
    case WRDE_SYNTAX :
        fprintf(stderr, "Erreur de syntaxe \n"); break;
    default :
        break;
    }
}

    #define LG_LIGNE    256

    int
main (void)
{
    char      ligne[LG_LIGNE];
    wordexp_t mots;
    int       erreur;
    pid_t     pid;
```

```
    while (1) {
        /* Lecture de la commande */
        fprintf(stdout, "-> ");
        if (fgets(ligne, LG_LIGNE, stdin) == NULL)
            break;
        if (strlen(ligne) == 0)
            continue;
        if (ligne[strlen(ligne) - 1] == '\n')
            ligne[strlen(ligne) - 1] = '\0';
        /* Analyse par wordexp() */
        if ((erreur = wordexp(ligne, & mots, WRDE_SHOWERR)) != 0) {
            affiche_erreur(erreur);
            goto fin_boucle;
        }
        if (mots.we_wordc == 0)
            goto fin_boucle;
        /* Traitement commande interne set */
        if (strcmp(mots.we_wordv [0], "set") == 0) {
            if (mots.we_wordc != 3) {
                fprintf(stderr, "syntaxe : set variable valeur \n");
                goto fin_boucle;
            }
            if (setenv(mots.we_wordv[1], mots.we_wordv[2], 1) < 0)
                perror("");
            goto fin_boucle;
        }
        /* Appel commande externe par un processus fils */
        if ((pid = fork ()) < 0) {
            perror("fork");
            goto fin_boucle;
        }
        if (pid == 0) {
            execvp(mots.we_wordv[0], mots.we_wordv);
            perror(mots.we_wordv[0]);
            exit(EXIT_FAILURE);
        } else {
            wait(NULL);
        }
        fin_boucle :
            wordfree(& mots);
    }
    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

Ce petit programme est simpliste, mais il est déjà étonnamment puissant :

```
$ ./exemple_wordexp
-> set VAR ETERNITE
-> echo ${VAR#*T}
ERNITE
-> ls ~ftp
bin  etc  lib  pub
```

```
-> set X 1
-> set Y $(($X + 2))
-> echo $Y
3
-> echo $(($Y * 25))
75
-> set DATE `date "+%d_%m_%Y"`
-> echo $DATE
05_01_2005
->  (Contrôle-D)
$
```

Bien entendu, nous sommes loin de la réalisation d'un véritable shell, capable d'interpréter les caractères spéciaux de redirection (<, ou >), les lancements de commandes en arrière-plan (&), etc. Malgré tout, nous voyons qu'avec quelques lignes de code il est déjà possible d'utiliser facilement la puissance des fonctions `wordexp()` et `wordfree()` de la bibliothèque C. Répétons qu'il faut être très prudent avec la substitution de commande, qui est un moyen très efficace pour introduire un trou de sécurité dans un programme Set-UID ou dans un démon. On privilégiera donc systématiquement l'option `WRDE_NOCMD`, à moins d'avoir vraiment besoin de cette fonctionnalité.

Descente récursive de répertoires

Pour l'instant, nous avons observé le moyen d'accéder au contenu d'un unique répertoire avec la fonction `scandir()`. Il est parfois nécessaire de descendre récursivement une arborescence en explorant tous ses sous-répertoires. Ceci peut se réaliser à l'aide de la commande **`ftw()`** ou de son dérivé **`nftw()`**, déclarées toutes deux dans `<ftw.h>`. Leurs prototypes sont :

```
int ftw (const char * depart,
         int (* fonction) (const char * nom,
                           const struct stat * etat,
                           int attributs),
         int profondeur);
```

Pour pouvoir utiliser `nftw()`, il faut définir la constante symbolique `_XOPEN_SOURCE` et lui donner la valeur `500` avant d'inclure `<ftw.h>`.

```
int nftw (const char * depart,
          int (* fonction) (const char * nom,
                            const struct stat * etat,
                            int attributs,
                            struct FTW * status),
          int profondeur,
          int options);
```

Ces deux fonctions partent du répertoire dont le chemin leur est fourni en premier argument. Elles parcourent son contenu en invoquant la fonction fournie en second argument pour chaque point d'entrée du répertoire. Ensuite, elles descendent récursivement dans toute l'arborescence. Une fois arrivées à la profondeur indiquée en troisième argument, ces fonctions devront refermer des descripteurs pour les réemployer à nouveau. Le nombre total de descripteurs disponibles simultanément pour un processus est en effet limité. Il y a deux différences entre ces deux fonctions : la première tient à un argument supplémentaire dans la

routine invoquée pour chaque entrée des répertoires, la seconde réside dans le quatrième argument de `nftw()`, qui permet de préciser son comportement.

Les routines appelées pour chaque entrée d'un répertoire reçoivent tout d'abord le nom de cet élément. Leur second argument est une structure `stat` que nous étudierons en détail dans le prochain chapitre, et qui contient diverses informations comme les dernières dates de modification ou d'accès, le numéro d'i-nœud, la taille du fichier, etc. Le troisième argument est un indicateur du type d'entrée, qui peut prendre l'une des valeurs suivantes :

Nom	Signification
`FTW_D`	L'élément est un répertoire.
`FTW_DNR`	L'élément est un répertoire dont on ne pourra pas lire le contenu.
`FTW_DP`	L'élément est un répertoire dont on a visité tous les sous-répertoires. Ceci n'est défini qu'avec `nftw()`, lorsque l'option `FTW_DEPTH` est utilisée.
`FTW_F`	L'élément est un fichier. Il faut toutefois se méfier car `ftw()` considère comme fichier tout ce qui n'est pas un répertoire.
`FTW_NS`	L'appel système `stat()` a échoué, le second argument de la routine n'est pas valide. Ce cas ne devrait normalement jamais se produire.
`FTW_SL`	L'élément est un lien symbolique. Comme `ftw()` suit les liens symboliques, ceci ne peut apparaître que si le lien pointe vers une destination inexistante. Par contre, pour `nftw()`, cet attribut apparaît si l'option `FTW_PHYS` est utilisée.
`FTW_SLN`	L'élément est un lien symbolique pointant vers une destination inexistante. Cet argument n'apparaît qu'avec `nftw()`.

La fonction invoquée lors de la descente récursive de `nftw()` reçoit donc un quatrième argument se présentant sous la forme d'une structure contenant les membres suivants :

Nom	Type	Signification
`base`	`int`	Il s'agit de la taille de la partie nom du fichier reçu en premier argument. Le reste de la chaîne est le chemin d'accès au fichier.
`level`	`int`	Il s'agit de la profondeur d'exploration de l'arborescence. La profondeur du répertoire de départ vaut 0.

Les options supplémentaires que propose `nftw()` sont les suivantes :

Nom	Rôle
`FTW_CHDIR`	Le processus change son répertoire de travail pour aller dans le répertoire exploré, avant d'appeler la routine fournie en second argument.
`FTW_DEPTH`	L'exploration se fait en profondeur d'abord, en descendant au plus bas avant de remonter dans les répertoires. Les répertoires seront alors détectés après leurs sous-répertoires (on recevra l'attribut `FTW_DP` et non `FTW_D`). Cette option permet de vider récursivement une arborescence à la manière de `rm -r`.
`FTW_MOUNT`	La fonction `nftw()` se limitera aux répertoires se trouvant sur le même système de fichiers que le répertoire de départ.
`FTW_PHYS`	Ne pas suivre les liens symboliques. La routine de l'utilisateur sera invoquée avec l'attribut `F_SL`. Si le lien pointe sur une destination inexistante, l'attribut `F_SLN` sera alors utilisé.

Si la fonction appelée pour un élément renvoie une valeur non nulle, `ftw()` arrête son exploration, libère les structures de données dynamiques qu'elle utilisait, et renvoie cette valeur. Sinon, elle se terminera lorsque tout le parcours sera fini, et renverra 0.

L'exemple ci-dessous est simplement un effacement récursif d'une arborescence. On prend garde à effacer les liens symboliques sans les suivre et à descendre jusqu'au bout des sous-répertoires avant de commencer à les vider.

exemple_nftw.c :

```
    #define _XOPEN_SOURCE 500
    #include <ftw.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
routine (const char * nom, const struct stat * etat,
         int attribut, struct FTW * status)
{
    if (attribut == FTW_DP)
        return rmdir(nom);
    return unlink(nom);
}

    int
main (int argc, char * argv[])
{
    int  i;
    for (i = 1; i < argc; i ++)
        if (nftw(argv[i], routine, 32,
                         FTW_DEPTH | FTW_PHYS | FTW_MOUNT) != 0)
            perror(argv[i]);

    return EXIT_SUCCESS;
}
```

Nous allons créer quelques fichiers et sous-répertoires pour pouvoir les supprimer par la suite :

```
$ mkdir ~/tmp/rep1
$ touch ~/tmp/rep1/fic1
$ touch ~/tmp/rep1/fic2
$ mkdir ~/tmp/rep1/rep1-1
$ touch ~/tmp/rep1/rep1-1/fic1
$ touch ~/tmp/rep1/rep1-1/fic2
$ ./exemple_nftw ~/tmp/rep1 /etc
/etc : Permission non accordée
$ cd ~/tmp
$ ls rep*
ls: rep*: Aucun fichier ou répertoire de ce type
$
```

Conclusion

Nous avons vu dans ce chapitre l'essentiel des fonctions permettant de travailler au niveau d'un répertoire, que ce soit pour en lire le contenu, créer des sous-répertoires, effacer ou déplacer des fichiers.

Les fonctions de mises en correspondance que nous avons étudiées pour rechercher des noms de fichiers sont très performantes, et permettent d'ajouter facilement à une application une interface puissante avec le système.

Pour avoir plus de détails sur la syntaxe des commandes arithmétiques du shell ou de la substitution des variables, on pourra se reporter par exemple à [NEWHAM 1995] *Le shell Bash*.

21

Attributs des fichiers

Pour le moment nous avons étudié les fichiers sous l'angle de leur contenu, des moyens d'y accéder et d'un point d'entrée dans un répertoire. Les fichiers existent pourtant également en tant qu'entité propre sur le disque, et c'est sous ce point de vue que nous allons les observer dans ce chapitre.

Nous examinerons tout d'abord les différentes informations que le système peut nous fournir sur un fichier, puis nous nous intéresserons successivement à tout ce qui concerne la taille du fichier, ses permissions d'accès, ses propriétaire et groupe, ainsi que les divers horodatages qui lui sont associés.

Informations associées à un fichier

Les informations que nous traitons dans ce chapitre sont indépendantes du contenu et du nom du fichier. Comme nous le verrons plus loin, un même fichier peut avoir plusieurs noms, dans un ou plusieurs répertoires. Pourtant, toutes les représentations de ce fichier partagent un certain nombre d'informations communes. Ces données peuvent être obtenues avec les appels-système **stat()**, **fstat()** ou **lstat()**. Tous trois fournissent leurs résultats dans une structure `stat`, définie dans `<sys/stat.h>`, que nous avons déjà rencontrée dans le chapitre précédent, à propos de `ftw()`. Cette structure renfermant en effet toutes les caractéristiques principales d'un fichier, on la retrouve très souvent.

La structure **stat** est définie par SUSv3 comme contenant les membres suivants :

Nom	Type	Signification
st_mode	mode_t	Ce champ contient les permissions d'accès au fichier ainsi que le type de ce dernier (répertoire, socket, fichier normal…). Les autorisations d'accès peuvent être modifiées avec l'appel-système chmod(). Pour déterminer le type du fichier, il existe des macros décrites plus bas.
st_ino	ino_t	La norme SUSv3 parle de numéro de référence du fichier. Il s'agit d'un identifiant unique permettant d'accéder au contenu du fichier. En pratique, sous Linux comme avec la majorité des Unix, on le nomme plutôt numéro d'i-nœud. Ce numéro est unique au sein d'un même système de fichiers.
st_dev	dev_t	Ce membre comprend le numéro du périphérique qui contient le système de fichiers auquel se rapporte le numéro d'i-nœud. Le couple st_ino et st_dev permet de définir de manière unique un fichier. La valeur st_dev n'est pas obligatoirement conservée entre deux redémarrages de la machine. Elle peut par exemple dépendre de l'ordre de détection des disques. On ne doit donc pas considérer qu'elle a une durée de vie plus longue que celle de l'exécution d'un processus.
st_nlink	nlink_t	Un fichier pouvant avoir plusieurs noms, ce champ en conserve le nombre. Il s'agit donc du nombre de liens physiques sur l'i-nœud. Lors d'un appel-système unlink(), cette valeur est décrémentée. Le fichier n'est véritablement supprimé que lorsque st_nlink arrive à zéro.
st_uid	uid_t	Ce champ contient l'UID du propriétaire du fichier. Il n'y a qu'un seul propriétaire pour un fichier, même si celui-ci dispose de plusieurs noms. Ce champ peut être modifié par l'appel-système chown().
st_gid	gid_t	Comme st_uid, ce membre identifie l'appartenance du fichier, mais cette fois-ci à un groupe. La valeur est modifiée également par l'appel-système chown().
st_size	off_t	La taille du fichier est ici mesurée en octets. Elle n'a de véritable signification que pour les fichiers normaux, pas pour les liens symboliques ni pour les fichiers spéciaux de périphérique.
st_atime	time_t	Ce membre contient la date du dernier accès au fichier. Elle est mise à jour lors de toute lecture ou écriture du contenu du fichier.
st_ctime	time_t	La date de changement du status du fichier est mise à jour à chaque consultation ou modification du contenu du fichier, mais également lors de la modification de ses caractéristiques (avec chmod(), chown()…).
st_mtime	time_t	Cette date est celle de la dernière modification du contenu du fichier. Elle n'est pas affectée par les changements de propriétaire, de permissions…
st_rdev	dev_t	Pour un fichier spécial représentant un périphérique, il s'agit des numéros d'identification majeur et mineur. Le numéro majeur est contenu dans le poids fort, le mineur dans le poids faible.
st_blksize	long	Il s'agit de la taille de bloc la mieux adaptée pour les entrées-sorties sur ce fichier. Elle est mesurée en octets. Cette valeur est très utile, nous l'avons vu au chapitre 18, lorsqu'on désire configurer la taille d'un buffer de sortie avec setvbuf(). Nous sommes assuré en utilisant un buffer dont la taille est un multiple de st_blksize d'avoir des entrées-sorties par flux optimales pour ce système de fichiers.
st_blocks	long	Cette valeur représente la taille effectivement allouée pour le fichier, telle qu'elle est mesurée par l'utilitaire du. Ce champ est évalué en nombre de blocs, mais la taille même des blocs n'est pas disponible de manière portable. On évitera d'utiliser ce membre.

Les prototypes des fonctions de la famille `stat()` sont déclarés dans `<unistd.h>`.

```
int stat (const char * nom_fichier, struct stat * infos);
int fstat (int descripteur, struct stat * infos);
int lstat (const char * nom_fichier, struct stat * infos);
```

La fonction `stat()` prend en premier argument un nom de fichier et remplit la structure `stat` sur laquelle on lui a transmis un pointeur en seconde position. Pour accéder aux informations concernant un fichier, il faut simplement avoir un droit de parcours (exécution) dans le répertoire le contenant, ainsi que dans les répertoires parents.

L'appel-système `lstat()` fonctionne comme `stat()`, mais lorsque le nom correspond à un lien symbolique, il fournit les informations concernant le lien lui-même et pas celles correspondant au fichier visé par le lien.

Enfin, `fstat()` utilise un descripteur de fichier déjà ouvert, ce qui peut permettre par exemple de vérifier le type de descripteur associé aux flux d'entrée ou de sortie standard (`STDIN_FILENO`, `STDOUT_FILENO`).

Pour vérifier le type d'un fichier, il faut utiliser une macro qui prend en argument le champ `st_mode` de la structure `stat`. Ces macros sont définies par Posix et prennent une valeur vraie si le fichier correspond au type indiqué.

Macro	Signification
`S_ISBLK (infos -> st_mode)`	Fichier spécial de périphérique en mode bloc
`S_ISCHR (infos -> st_mode)`	Fichier spécial de périphérique en mode caractère
`S_ISDIR (infos -> st_mode)`	Répertoire
`S_ISFIFO (infos -> st_mode)`	FIFO (tube nommé)
`S_ISLNK (infos -> st_mode)`	Lien symbolique
`S_ISREG (infos -> st_mode)`	Fichier régulier
`S_ISSOCK (infos -> st_mode)`	Socket

Pour connaître les autorisations d'accès du fichier, on prend des constantes symboliques qu'on compare en utilisant un ET binaire au champ `st_mode` de la structure `stat`. Ces constantes ont été présentées dans le chapitre 19, lors de la description du troisième argument de `open()`. Il s'agit des valeurs `S_IRUSR`, `S_IXGRP`, etc. On peut éventuellement se permettre, au risque d'une portabilité légèrement amoindrie, de consulter directement la valeur `st_mode`, en effectuant un ET binaire avec le masque octal 07777, comme nous l'avions signalé à propos de `open()`.

Rappelons qu'un fichier ne possédant pas l'autorisation d'exécution pour son groupe `S_IXGRP`, mais ayant par contre l'attribut Set-GID `I_SGID`, est en réalité un fichier sur lequel un verrouillage strict s'applique, comme nous en avons vu des exemples à la fin du chapitre 19.

Le programme suivant permet de connaître le type et les autorisations d'accès à un fichier. Si on lui transmet un ou plusieurs noms en arguments, il utilise `stat()` pour obtenir les informations. Si on ne lui envoie rien, il invoque `fstat()` pour afficher les données correspondant à ses flux d'entrée et de sortie standard.

exemple_stat.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>

    void
affiche_status (struct stat * status)
{
    if (S_ISBLK(status->st_mode))
        fprintf(stderr, "bloc ");
    else if (S_ISCHR(status->st_mode))
        fprintf(stderr, "caractère ");
    else if (S_ISDIR(status->st_mode))
        fprintf(stderr, "répertoire ");
    else if (S_ISFIFO(status->st_mode))
        fprintf(stderr, "fifo ");
    else if (S_ISLNK(status->st_mode))
        fprintf(stderr, "lien ");
    else if (S_ISREG(status->st_mode))
        fprintf(stderr, "fichier ");
    else if (S_ISSOCK(status->st_mode))
        fprintf(stderr, "socket ");
    fprintf(stderr, "u:");
    fprintf(stderr, status->st_mode & S_IRUSR ? "r" : "-");
    fprintf(stderr, status->st_mode & S_IWUSR ? "w" : "-");
    fprintf(stderr, status->st_mode & S_IXUSR ? "x" : "-");
    fprintf(stderr, " g:");
    fprintf(stderr, status->st_mode & S_IRGRP ? "r" : "-");
    fprintf(stderr, status->st_mode & S_IWGRP ? "w" : "-");
    fprintf(stderr, status->st_mode & S_IXGRP ? "x" : "-");
    fprintf(stderr, " o:");
    fprintf(stderr, status->st_mode & S_IROTH ? "r" : "-");
    fprintf(stderr, status->st_mode & S_IWOTH ? "w" : "-");
    fprintf(stderr, status->st_mode & S_IXOTH ? "x" : "-");
    fprintf(stderr, "\n");
}

int
main (int argc, char * argv[])
{
    struct stat status;
    int         i;

    if (argc == 1) {
        fprintf(stderr, "stdin : ");
        if (fstat(STDIN_FILENO, & status) < 0)
            perror("");
        else
            affiche_status(& status);
        fprintf(stderr, "stdout : ");
```

```
        if (fstat(STDOUT_FILENO, & status) < 0)
            perror("");
        else
            affiche_status(& status);
    } else {
        for (i = 1; i < argc ; i ++) {
            fprintf(stderr, "%s : ", argv[i]);
            if (stat(argv[i], & status) < 0)
                perror("");
            else
                affiche_status(& status);
        }
    }
    return EXIT_SUCCESS;
}
```

Lors de l'exécution de ce programme, nous essayons d'examiner les principaux types de fichiers, périphériques de type bloc ou caractère, répertoires, fichiers normaux et FIFO obtenus grâce à une redirection du shell.

```
$ ./exemple_stat /dev/hda1 /dev/ttyS2 /etc/passwd
/dev/hda1 : bloc u:rw- g:rw- o:---
/dev/ttyS2 : caractère u:rw- g:--- o:---
/etc/passwd : fichier u:rw- g:r-- o:r--
$ ./exemple_stat /etc
/etc : répertoire u:rwx g:r-x o:r-x
$ ./exemple_stat | cat
stdin : caractère u:rw- g:-w- o:---
stdout : fifo u:rw- g:--- o:---
$ cat /dev/null | ./exemple_stat
stdin : fifo u:rw- g:--- o:---
stdout : caractère u:rw- g:-w- o:---
$
```

Autorisation d'accès

Pour modifier les autorisations d'accès à un fichier, on utilise l'un des appels-système **`chmod()`** ou **`fchmod()`** déclarés dans `<sys/stat.h>` :

```
int chmod (const char * nom_fichier, mode_t mode);
int fchmod (int descripteur, mode_t mode);
```

On voit que `fchmod()` agit directement sur un descripteur de fichier déjà ouvert alors que `chmod()` travaille sur un nom de fichier. Pour être autorisé à changer les autorisations associées à un fichier, il faut que l'UID effectif du processus appelant soit égal à 0 (*root*) ou à celui du propriétaire du fichier (indiqué dans le champ `st_uid` de la structure `stat`). Si toutefois le GID du fichier (champ `st_gid`) n'est égal à aucun des groupes auxquels appartient le processus appelant, et si l'UID effectif de ce dernier n'est pas nul, le bit `S_ISGID` (Set-GID) sera silencieusement effacé.

Lorsqu'un processus accède véritablement à un fichier grâce aux appels-système `open()` ou `execve()`, l'UID pris en compte pour vérifier les autorisations est l'UID effectif du processus et pas son UID réel. Cela pose un problème pour les processus Set-UID. Supposons que nous

écrivons un programme pilotant une interface spécifique personnalisée (automate industriel, instrument de mesure scientifique…). Cette application, pour dialoguer avec notre dispositif, doit employer des appels-système privilégiés. Pour avoir le droit d'exploiter ces appels et pour pouvoir être employée par n'importe quel utilisateur, l'application doit être installée Set-UID *root*. Notre processus dispose donc de la toute-puissance de *root*. Toutefois, nous désirons également que l'utilisateur puisse sauvegarder des données dans ses propres fichiers. On ne peut pas utiliser directement l'appel `open()` car, l'UID effectif étant celui de *root*, notre utilisateur pourrait écraser n'importe quel fichier. Il est possible d'abandonner temporairement nos privilèges, comme nous l'avons étudié dans le chapitre 2, mais c'est fastidieux si on alterne régulièrement des entrées-sorties avec `inb()`-`outb()` et des ouvertures de fichiers.

Le même problème se pose avec des applications comme l'utilitaire de communication `minicom`, qui doit être Set-UID (ou au moins Set-GID) pour avoir le droit d'accéder au périphérique de liaison série, et qui permet à l'utilisateur d'enregistrer sa configuration ou l'ensemble de sa session dans un fichier.

Il existe donc un appel-système nommé **`access()`** qui permet de vérifier si un processus peut exécuter ou non un accès particulier à un fichier en se fondant sur son UID réel (celui de l'utilisateur qui a lancé le processus). Il est déclaré dans `<unistd.h>` :

```
int access (const char * nom_fichier, int mode);
```

Le mode qu'on transmet en second argument correspond à l'utilisation qu'on désire faire du fichier. Il existe quatre constantes symboliques :

Nom	Signification
`F_OK`	Le fichier existe-t-il ?
`R_OK`	Puis-je lire le contenu du fichier ?
`W_OK`	Puis-je écrire dans le fichier ?
`X_OK`	Puis-je exécuter le fichier ?

Attention

La vérification n'a lieu qu'en ce qui concerne les bits d'autorisation du fichier, le test d'exécution peut très bien réussir alors que `execve()` échouera si le fichier n'est pas dans un format exécutable correct.

La valeur renvoyée par cet appel-système est nulle si l'accès est autorisé, et vaut –1 sinon. La variable globale `errno` est dans ce cas remplie.

On emploie donc `access()` immédiatement avant l'appel `open()` ou `execve()` correspondant. Il faut être conscient du risque potentiel concernant la sécurité, car il existe un petit délai entre la vérification des autorisations avec l'UID réel et l'ouverture du fichier avec l'UID effectif. Un utilisateur malintentionné pourrait profiter de ce délai pour supprimer le fichier banal qu'il proposait de modifier et le remplacer par un lien matériel vers un fichier système (`/etc/passwd` par exemple) que l'UID effectif du processus pourra ouvrir. Pour éviter ce genre de désagrément, on préférera autant que possible perdre temporairement nos privilèges pour retrouver l'identité effective de l'utilisateur ayant lancé le programme, en employant les appels-système `setreuid()` ou `setresuid()`.

Propriétaire et groupe d'un fichier

Lorsqu'un processus Set-UID vérifie avec `access()` s'il peut écrire ou créer un fichier et qu'il le crée effectivement, avec `open()` ou `creat()`, ce nouveau fichier possède les UID et GID effectifs du processus appelant. Il lui faut donc modifier les appartenances du nouveau fichier pour les faire correspondre à celles de l'utilisateur.

Seul un processus ayant un UID effectif nul ou la capacité `CAP_CHOWN` peut modifier le propriétaire d'un fichier. Par contre, le propriétaire d'un fichier peut l'affecter à n'importe quel groupe auquel il appartient lui-même. Les identités du propriétaire et du groupe sont communes à toutes les occurrences du fichier à travers ses différents noms (*via* des liens physiques).

Pour modifier l'UID ou le GID d'un fichier, on emploie les appels-système **`chown()`**, **`fchown()`** et **`lchown()`**, déclarés dans `<unistd.h>` :

```
int chown  (const char * nom_fichier, uid_t proprietaire, gid_t groupe);
int fchown (int descripteur_fichier,  uid_t proprietaire, gid_t groupe);
int lchown (const char * nom_fichier, uid_t proprietaire, gid_t groupe);
```

Les appels-système `chown()` et `lchown()` modifient l'appartenance d'un fichier dont le nom leur est fourni en premier argument. La différence concerne les liens symboliques ; `chown()` modifie l'appartenance du fichier visé par le lien alors que `lchown()` s'applique au lien lui-même – ce qui ne présente pas beaucoup d'intérêt. De son côté, `fchown()` agit sur le descripteur d'un fichier déjà ouvert. Si l'un des UID ou GID indiqués vaut –1, cette valeur n'est pas changée.

Lors de la modification du propriétaire, le bit Set-UID éventuel est effacé. Lors de la modification du groupe, le bit Set-GID est effacé si ce fichier possède également le bit d'exécution pour son groupe (sinon c'est un fichier avec un verrouillage strict).

On reprend l'exemple d'un logiciel de communication comme `minicom` et d'une séquence `access()` suivie de `open()` comme nous l'avons décrit ci-dessus pour créer un fichier d'enregistrement. La modification du propriétaire de ce fichier nécessiterait en théorie que l'application soit Set-UID *root*. Toutefois, il existe d'autres possibilités, notamment on peut utiliser un exécutable Set-GID appartenant à un groupe autorisé à accéder aux ports de communication (*uucp* par exemple). La modification de l'appartenance du nouveau fichier est alors restreinte à celle de son groupe, ce qui peut être envisagé avec n'importe quel UID effectif. Il vaut mieux, pour des raisons de sécurité, employer dans ce cas `fchown()` directement sur le descripteur du fichier qu'on vient de créer plutôt que `chown()` sur son nom, car l'utilisateur pourrait à nouveau exploiter le délai entre la création du fichier et la modification de son appartenance pour l'effacer et le remplacer par un lien matériel sur un fichier système.

Taille du fichier

La taille d'un fichier est indiquée par le champ `st_size` de la structure `stat`. Nous reprenons le programme `exemple_stat.c` et nous le modifions pour obtenir `exemple_tailles.c` :

- On remplace l'affichage des modes `status->st_mode` par l'affichage de la taille `status->st_size`.
- On remplace l'appel `stat()` par `lstat()` pour ne pas suivre les liens symboliques mais s'intéresser au lien lui-même.

Le programme ainsi obtenu nous permet de formuler plusieurs observations :

```
$ ./exemple_tailles /dev/hda /dev/ttyS0
/dev/hda : bloc 0
/dev/ttyS0 : caractère 0
$
```

Les fichiers spéciaux (en mode caractère ou bloc) ont une taille nulle. En réalité, il s'agit simplement d'indicateurs pour le noyau. Ils occupent une entrée de répertoire, mais pas d'autre place sur le disque. Nous reviendrons sur ces fichiers plus loin.

```
$ ./exemple_tailles /etc /usr /dev
/etc : répertoire 8192
/usr : répertoire 4096
/dev : répertoire 118784
$ mkdir vide
$ ./exemple_tailles vide
vide : répertoire 4096
$ rmdir vide
$
```

Un répertoire occupe une taille (multiple de 4 096 sur ce système de fichiers) correspondant à son contenu, c'est-à-dire les noms des fichiers et les pointeurs vers les i-nœuds.

```
$ ls -l /etc/services
-rw-r—r--  1 root root     11279   Nov 10  11:34   /etc/services
$ ln -sf /etc/services .
$ ./exemple_tailles /etc/services services
/etc/services : fichier 11279
services : lien 13
$ rm services
$
```

Un fichier normal occupe la taille nécessaire pour stocker son contenu. Un lien symbolique n'emploie que la taille indispensable pour enregistrer le nom du fichier vers lequel il pointe (en l'occurrence « `/etc/services` » comporte 13 caractères).

```
$ cat /dev/null | ./exemple_tailles | cat
stdin : fifo 0
stdout : fifo 0
$
```

Les tubes et les FIFO sont des structures particulières n'ayant pas de taille attribuée (bien qu'elles aient toutefois une dimension maximale).

Remarquons bien que les données fournies par le membre `st_size` de `stat` correspond à la taille des données contenues dans un fichier, et pas forcément à son occupation sur le disque. En voici un exemple :

```
$ ./exemple_tailles /etc/services
/etc/services : fichier 11279
$ du -b /etc/services
12288   /etc/services
$
```

L'utilitaire `du` calcule en effet la place occupée en prenant en compte la taille des blocs du système de fichiers et le nombre de blocs employés.

Lorsqu'on désire augmenter la dimension d'un fichier, on utilise simplement les fonctions d'écriture. Pour diminuer sa taille, le travail est plus compliqué : il faut procéder en réalisant une copie partielle du fichier original, qu'on renommera ensuite. Une autre possibilité consiste à utiliser des fonctions **`truncate()`** ou **`ftruncate()`**, déclarées dans `<unistd.h>` :

```
int truncate  (const char * nom_fichier, off_t longueur);
int ftruncate (int descripteur_fichier,  off_t longueur);
```

Dans le cas d'une réduction de taille, les données supplémentaires se trouvant en fin de fichier sont simplement éliminées. Si la longueur demandée est plus grande que la taille actuelle du fichier, la zone intermédiaire est remplie de zéros. Comme nous l'avions déjà observé dans le chapitre 19, ces zéros sont autant que possible des trous dans le fichier.

exemple_truncate.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (int argc, char * argv[])
{
    long longueur;
    if ((argc != 3) || (sscanf(argv[2], "%ld", & longueur) != 1)) {
        fprintf(stderr, "Syntaxe : %s fichier longueur \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (truncate(argv[1], longueur) < 0)
        perror(argv[1]);
    return EXIT_SUCCESS;
}
```

Nous utilisons le programme `exemple_getchar` du chapitre 10 pour examiner le contenu d'un fichier que nous fabriquons et dont nous modifions la taille.

```
$ echo -n "abcdefghijklmnopqrstuvwxyz" > essai.truncate
$ ls -l essai.truncate
-rw-rw-r--   1 ccb      ccb        26 Dec 30 23:34 essai.truncate
$ ./exemple_truncate essai.truncate 10
$ ls -l essai.truncate
-rw-rw-r--   1 ccb      ccb        10 Dec 30 23:34 essai.truncate
$ ../10/exemple_getchar < essai.truncate
00000000  61 62 63 64 65 66 67 68-69 6A                    abcdefghi
$ ./exemple_truncate essai.truncate 20
$ ls -l essai.truncate
-rw-rw-r--   1 ccb      ccb        20 Dec 30 23:34 essai.truncate
$ ../10/exemple_getchar < essai.truncate
00000000  61 62 63 64 65 66 67 68-69 6A 00 00 00 00 00 00  abcdefghij
00000010  00 00 00 00
$ rm essai.truncate
$
```

`ftruncate()` peut être très utile pour fixer précisément la longueur d'un fichier qu'on désire projeter en mémoire. Cela s'applique principalement aux applications qui veulent projeter simultanément plusieurs zones du même fichier, en ajoutant ainsi des blocs complets en fin de fichier.

Horodatages d'un fichier

Nous avons observé dans la structure `stat` que trois dates sont associées à un fichier :

- `st_atime`, la date du dernier accès au contenu du fichier, en lecture ou en écriture.
- `st_mtime`, la date de dernière modification du contenu du fichier avec une primitive d'écriture.
- `st_ctime`, la date de dernière modification de la structure `stat` associée au fichier, ce qui inclut le changement de propriétaire, de mode…

Le type utilisé pour enregistrer ces dates est `time_t`, qui s'exprime en secondes écoulées depuis le 1er janvier 1970. Nous reviendrons sur ces types de données dans le chapitre 25.

Les horodatages sont mis à jour automatiquement par le noyau, mais on peut avoir besoin pour de nombreuses raisons de modifier les dates `st_atime` ou `st_mtime`. Le champ `st_ctime` ne peut être mis à jour que par le noyau. Les appels **`utime()`** et **`utimes()`** sont déclarés respectivement dans `<utime.h>` et `<sys/time.h>`. Ils servent tous deux à mettre à jour les dates `st_atime` et `st_mtime`, mais `utimes()` permet d'accéder à une précision de l'ordre de la microseconde, alors que `utime()` est limité à la seconde près par le type `time_t`.

```
int utime  (const char * nom_fichier, struct utimbuf * dates);
int utimes (const char * nom_fichier, struct timeval dates [2]);
```

La structure `utimbuf` contient les champs suivants :

Nom	Type	Signification
`actime`	`time_t`	Date du dernier accès au contenu du fichier
`modtime`	`time_t`	Date de dernière modification du contenu du fichier

Pour l'appel `utimes()`, il faut passer un tableau de deux structures, la première correspondant au dernier accès, et la seconde à la dernière modification. Les membres des structures `timeval` sont :

Nom	Type	Signification
`tv_sec`	`long`	Nombre de secondes écoulées depuis le 1er janvier 1970
`tv_usec`	`long`	Nombre de microsecondes (0 à 999 999)

Linux ne permet pas de mémoriser une telle précision, et `utimes()` est ainsi implémenté en remplissant simplement les champs d'une structure `timebuf` avant d'appeler `utime()`.

Si le pointeur passé en second argument de `utime()` ou de `utimes()` est `NULL`, les dates sont mises à jour au moment de l'appel.

Liens physiques

Nous avons déjà évoqué à plusieurs reprises les liens physiques ou liens matériels (*hard links*). Cette notion de lien est un peu trompeuse, car elle suggère une entité inamovible, originelle, à laquelle on rattache des avatars de moindre importance. Si cette image peut s'appliquer aux liens symboliques que nous verrons dans le prochain paragraphe, elle est totalement erronée dans le cas des liens physiques.

Lorsqu'on crée un lien physique sur un fichier, on ajoute simplement un nouveau nom dans le système de fichiers, qui pointe vers le même i-nœud que le nom original. Néanmoins, il n'y a plus aucun moyen de distinguer le nom qui était le premier et celui qui a été créé ensuite. Les deux noms sont traités avec égalité par le système. En fait, il faut considérer que tout nom présent dans le système de fichiers est un lien physique vers le contenu même du fichier.

Il n'est pas possible de créer dans un répertoire un nom lié à un fichier se trouvant sur un autre système de fichiers. De plus, certains systèmes Unix, dont Linux, n'autorisent pas la création de liens physiques sur un répertoire. On pourrait en effet concevoir une boucle dans le système de fichiers (un répertoire contenant un sous-répertoire correspondant à son père), et le noyau n'est pas prêt à détecter de tels conflits (contrairement à ce qui se passe avec les liens symboliques).

Pour créer un lien matériel, il existe un appel-système nommé `link()`, décrit par Posix, et déclaré dans `<unistd.h>` :

```
int link (const char * nom_original, const char * nouveau_nom);
```

Cet appel-système établit donc un nouveau lien sur le fichier transmis en second argument, créant ainsi un nouveau nom dans le système de fichiers. Le champ `st_nlink` de la structure `stat` correspondant à l'i-nœud est incrémenté. Cette valeur est également visible dans la seconde colonne affichée lors d'un `ls -l`. Notons que le nouveau nom ne sera pas écrasé s'il existe déjà. Si on veut forcer la création, il faut l'effacer auparavant.

Nous comprenons à présent pourquoi l'appel-système d'effacement d'un nom de fichier se nomme `unlink()`, puisqu'il sert simplement à supprimer un lien physique sur le contenu du fichier et à décrémenter ainsi le champ `st_nlink`. Lorsque ce dernier arrive à zéro, l'espace occupé sur le disque est réellement libéré.

Il existe une application `/bin/ln` qui sert de frontal à l'appel-système `link()`. Lorsque nous ne précisons aucune option, cet utilitaire crée un lien physique. En voici un exemple d'utilisation :

```
$ ls -l exemple_tailles.c
-rw-rw-r--   1 ccb      ccb          1219 Dec 30 17:56 exemple_tailles.c
$ ln exemple_tailles.c deuxieme_nom.c
$ ls -l exemple_tailles.c deuxieme_nom.c
-rw-rw-r--   2 ccb      ccb          1219 Dec 30 17:56 deuxieme_nom.c
-rw-rw-r--   2 ccb      ccb          1219 Dec 30 17:56 exemple_tailles.c
$ rm deuxieme_nom.c
$ ls -l exemple_tailles.c
-rw-rw-r--   1 ccb      ccb          1219 Dec 30 17:56 exemple_tailles.c
$
```

Nous remarquons que le champ `st_nlink` de la structure `stat`, affiché en seconde colonne du résultat de `ls -l`, passe bien à 2, puis revient à 1 après la destruction de l'un des noms. Vérifions à présent que les liens matériels sont interdits sur les répertoires et entre plusieurs systèmes de fichiers différents :

```
$ ln /tmp .
ln: /tmp: hard link not allowed for directory
$ mount /mnt/dos
$ ln /mnt/dos/autoexec.bat .
ln: cannot create hard link `./autoexec.bat' to `/mnt/dos/autoexec.bat':
      Invalid cross-device link
$
```

Les liens physiques sont souvent utilisés pour donner plusieurs noms différents à la même application, de manière transparente vis-à-vis de l'utilisateur. Dans mon système actuel, je peux ainsi vérifier la présence de plusieurs liens physiques dans le répertoire `/bin` par exemple :

```
$ ls -l /bin |grep "r-x   2"
-rwxr-xr-x   2 root     root       150964 Jul  1  1999 gawk
-rwxr-xr-x   2 root     root       150964 Jul  1  1999 gawk-3.0.4
$ ls -l /bin |grep "r-x   3"
-rwxr-xr-x   3 root     root        50384 Mar 25  1999 gunzip
-rwxr-xr-x   3 root     root        50384 Mar 25  1999 gzip
-rwxr-xr-x   3 root     root        50384 Mar 25  1999 zcat
$
```

L'application `gzip` se présente ainsi sous trois noms différents. Lorsque le programme démarre, il analyse `argv[0]` dans la fonction `main()` pour savoir comment se comporter. On peut aussi utiliser l'option –`d` de `gzip` pour assurer une décompression, mais l'appel « `gzip -d fichier.gz` » est moins intuitif que « `gunzip fichier.gz` ». Lorsqu'on voit qu'un fichier dispose de plusieurs noms grâce à la seconde colonne de `ls -l`, il n'est toutefois pas possible de savoir immédiatement où ils se trouvent. La taille du fichier est un indice sérieux mais pas totalement sûr. On peut se servir de l'application `find` avec son option –`inum` pour comparer le numéro d'i-nœud. Ce dernier est affiché avec l'option –`i` de `ls`. En voici un exemple :

```
$ ls -i -l /usr/bin/gcc
  62459 -rwxr-xr-x   3 root  root    64604 Sep  8 23:11 /usr/bin/gcc
$
```

L'exécutable `gcc` a donc trois noms différents. Recherchons-les avec `find`, à partir de `/usr`, en utilisant l'option –`xdev` (inutile de parcourir les autres systèmes de fichiers, tous les liens physiques doivent résider sur le même) et –`inum` pour trouver les fichiers dont le numéro d'i-nœud soit 62459 :

```
$ find /usr -xdev -inum 62459
/usr/bin/i386-redhat-linux-gcc
/usr/bin/egcs
/usr/bin/gcc
$
```

Liens symboliques

Contrairement à leurs homonymes physiques, les liens symboliques (*soft links* ou *symbolic links*) sont soumis à un nombre moins important de contraintes. Ils sont également implémentés conceptuellement à un niveau plus élevé dans l'organisation du système de fichiers.

Un lien symbolique n'est rien de plus qu'un petit fichier de texte comprenant le chemin d'accès et le nom du fichier vers lequel il pointe. Le lien est également marqué par un type spécial – qu'on détermine grâce à la macro `S_ISLNK()` appliquée au champ `st_mode` de la structure `stat` – et qui permet au noyau de le reconnaître. Avec certains appels-système, le noyau agira alors sur le contenu du lien, en opérant sur le fichier visé, alors que d'autres primitives fonctionneront directement sur le lien symbolique lui-même.

Un lien symbolique est créé grâce à l'appel **`symlink()`**, déclaré dans `<unistd.h>` :

```
int symlink (const char * nom_original, const char * nouveau_nom);
```

On peut utiliser aussi l'utilitaire `/bin/ln` avec l'option `–s` pour créer un lien symbolique.

> **Attention**
> Il est tout à fait possible de créer un lien symbolique pointant vers un fichier inexistant. Le système indiquera une erreur lors de la tentative d'ouverture. De même, si le fichier visé est supprimé, les liens symboliques qui pointaient vers lui ne sont pas concernés.

En conséquence, on peut alors créer un lien symbolique entre différents systèmes de fichiers et créer des liens sur des répertoires. On peut aussi concevoir des boucles dans les liens, ce que le système détectera lors des tentatives d'accès.

Voici la création d'un lien symbolique normal et sa suppression :

```
$ ls -l Makefile
-rw-r--r--   1 ccb   ccb     203 Dec 30 23:22 Makefile
$ ln -s Makefile Makefile.2
$ ls -l Makefile*
-rw-r--r--   1 ccb   ccb     203 Dec 30 23:22 Makefile
lrwxrwxrwx   1 ccb   ccb       8 Jan  2 00:41 Makefile.2 -> Makefile
$ rm Makefile.2
$ ls -l Makefile*
-rw-r--r--   1 ccb   ccb     203 Dec 30 23:22 Makefile
$
```

La taille d'un lien symbolique est limitée à la longueur du chemin qu'il contient (toutefois l'allocation de l'espace sur le disque est assurée par blocs beaucoup plus gros). Voici un exemple de lien symbolique pointant vers un fichier inexistant :

```
$ ln -s /tmp/je_n_existe_pas ici
$ ls -l ici
lrwxrwxrwx   1 ccb  ccb    20 Jan  2 00:44 ici -> /tmp/je_n_existe_pas
$ cat ici
cat: ici: Aucun fichier ou répertoire de ce type
$ rm ici
$
```

Un lien symbolique peut contenir un chemin absolu depuis la racine du système de fichiers ou un chemin relatif à base de `./` ou `../`. Cette dernière solution est souvent préférable, surtout si le répertoire est susceptible d'être exporté par le système NFS pour être visible sur d'autres systèmes n'ayant pas la même arborescence. Notons que certains systèmes de fichiers (par exemple `msdos` ou `vfat`) ne permettent pas la création de liens symboliques.

Les permissions d'accès à un lien symbolique ne sont jamais prises en compte, aussi sont-elles fixées automatiquement à `rwxrwxrwx`. Le nom du propriétaire d'un lien symbolique n'a que rarement de l'intérêt. Le seul cas où cette information est utile est celui où le lien réside dans un répertoire public ayant son bit Sticky à 1, ce qui signifie que seul *root* ou le propriétaire d'un fichier peuvent l'effacer ou le modifier.

Les liens symboliques sont une méthode très commode pour configurer un système, particulièrement dans une arborescence de fichiers source. On utilise par exemple souvent des liens symboliques dans des applications portées sur plusieurs plates-formes, pour faire pointer un seul fichier `Makefile` au choix vers `Makefile.linux`, `Makefile.aix`, `Makefile.solaris`, `Makefile.hpux`, etc. De même, chaque équipe de développement a souvent une petite bibliothèque de fichiers source réutilisés dans plusieurs projets. Ces fichiers peuvent être conservés en un seul exemplaire, permettant ainsi une mise à jour automatique en cas de correction à apporter, tout en ayant des liens symboliques dans les arborescences de chaque projet qui les utilise.

Le lien symbolique présente l'avantage d'être une indirection explicite, facilement visible, au contraire des liens physiques. Ceci permet de changer l'emplacement réel d'un fichier, tout en le laissant accessible à partir d'un autre répertoire où il était placé historiquement. C'est le cas du répertoire `/usr/tmp` qu'on conserve pour des raisons historiques, mais qui est généralement lié symboliquement au répertoire `/var/tmp`. La partition `/usr` doit en effet pouvoir être montée en lecture seule (voire depuis un serveur NFS). Alors que `/var` est souvent une partition locale, comme `/tmp`, sur laquelle on peut éventuellement recréer le système de fichiers à chaque redémarrage de la machine. Les liens symboliques sont alors des outils précieux pour les administrateurs qui gèrent un parc de plusieurs machines hétérogènes utilisant la même arborescence `/usr` depuis un serveur NFS, puisqu'ils permettent d'employer un lien symbolique dans la partition commune, comme le lien `/usr/X11R6/lib/X11/XF86Config` vers un fichier dépendant de chaque machine `/etc/X11/XF86Config`.

Lorsqu'on utilise `open()` sur un lien symbolique, cet appel-système tente d'accéder au fichier visé. Pour connaître le véritable contenu du lien (le chemin vers lequel il pointe), il faut employer un appel-système différent, nommé **`readlink()`**, déclaré dans `<unistd.h>` :

```
int readlink (const char * nom_lien, char * buffer, size_t taille);
```

Cet appel-système recopie dans le buffer passé en deuxième argument le contenu du lien dont le nom est passé en première position. Il limite la longueur de la copie à la taille fournie en troisième argument, mais n'ajoute pas de caractère nul.

La valeur renvoyée vaut –1 en cas d'échec. Si `readlink()` réussit, il renvoie le nombre de caractères copiés. Si ce nombre est égal à la taille maximale, on recommencera donc l'appel avec un buffer plus grand, comme dans l'exemple suivant :

exemple_readlink.c :

```
#include <stdlib.h>
#include <unistd.h>
```

```
    void
lecture_contenu (const char * nom)
{
    char * buffer = NULL;
    char * nouveau = NULL;
    int    taille = 0;
    int    nb_copies;

    while (1) {
        taille += 16;
        if ((nouveau = realloc(buffer, taille)) == NULL) {
            perror(nom);
            break;
        }
        buffer = nouveau;
        if ((nb_copies = readlink(nom, buffer, taille - 1)) == -1) {
            perror(nom);
        if (nb_copies < taille - 1) {
            buffer[nb_copies] = '\0';
            fprintf(stdout, "%s : %s\n", nom, buffer);
            break;
        }
    }
    free(buffer);
}

    int
main (int argc, char * argv[])
{
    int     i;
    for (i = 1; i < argc; i ++)
        lecture_contenu(argv[i]);

    return EXIT_SUCCESS;
}
```

Dans l'exécution suivante, nous créons un lien dont le contenu est largement plus long que 16 caractères pour vérifier que notre routine fonctionne :

```
$ ln -s /etc/services .
$ ln -s /usr/X11R6/include/X11/bitmaps/escherknot .
$ ./exemple_readlink services escherknot
services : /etc/services
escherknot : /usr/X11R6/include/X11/bitmaps/escherknot
$ rm escherknot services
$
```

Il n'est pas toujours évident de savoir si un appel-système s'applique au lien symbolique lui-même ou à son contenu. En règle générale, le comportement des appels-système est dicté par le bon sens. En voici toutefois un récapitulatif rapide pour les principales primitives de traitement des fichiers :

Appel-système	concerne le lien lui-même	concerne le fichier visé
access		•
chdir		•
chmod		•
chown	jusqu'à Linux 2.0	depuis Linux 2.2
execve		•
lchown	•	
lstat	•	
open		•
readlink	•	
rename	•	
stat		•
truncate		•
unlink	•	
utime	•	
utimes	•	

Nœud générique du système de fichiers

Il existe, nous l'avons constaté dans la structure `stat`, plusieurs types de nœuds, qu'on peut rencontrer dans un système de fichiers. Tout d'abord, on trouve les fichiers réguliers qu'on crée avec l'appel-système `open()` ou `creat()` ; bien entendu, il existe également les répertoires, créés avec `mkdir()`, ainsi que les liens physiques et symboliques issus de `link()` et `symlink()`. Les sockets ne se trouvent généralement pas dans le système de fichiers[1], mais leurs descripteurs – fournis par l'appel-système `socket()` que nous analyserons dans le chapitre 32 – sont manipulés comme les descripteurs de fichiers et peuvent être transmis en argument de `fstat()`. On peut encore trouver trois types de nœuds : les files FIFO, les fichiers spéciaux de périphérique de type caractère, et ceux de type bloc. Pour créer ce genre de nœuds, on utilise l'appel-système **`mknod()`**, déclaré dans `<sys/stat.h>` :

```
int mknod (const char * nom, mode_t mode, dev_t periph);
```

Le premier argument de cet appel-système indique le nom du nœud à créer, et le mode précisé à la suite doit être l'une des constantes suivantes :

Nom	Signification
S_IFREG	Création d'un fichier régulier vide, équivalent d'un `open()` suivi d'un `close()`. Le troisième argument de `mknod()` est ignoré. Ce mode de fonctionnement ne nous intéressera pas ici.
S_IFIFO	Création d'une file FIFO. Ce type de fichier est habituellement créé à l'aide de la fonction de bibliothèque `mkfifo()`, que nous analyserons dans le chapitre 28. Il s'agit d'un moyen de communication entre processus. La création d'une FIFO avec `mknod()` ne nous intéressera pas non plus.

1. Dans certaines situations, une socket peut quand même avoir un nom dans le système de fichiers. C'est le cas par exemple de `/dev/log` ou de `/dev/printer`. Leur comportement dans ce cas s'apparente assez à celui d'une file FIFO.

Nom	Signification
S_IFBLK	Création d'un fichier spécial de périphérique de type bloc.
S_IFCHR	Création d'un fichier spécial de périphérique de type caractère.

Dans le cas d'un fichier spécial de périphérique (bloc ou caractère), le troisième argument indique au noyau le type du pilote de périphérique désiré. Cette valeur est composée de deux nombres, qu'on nomme numéro majeur et numéro mineur du périphérique. Le numéro majeur permet au noyau de déterminer quel pilote de périphérique est concerné lorsqu'on tente une ouverture, une lecture ou une écriture sur le nœud dont il est question. Le numéro mineur est réservé au pilote lui-même, pour pouvoir différencier plusieurs dispositifs matériels par exemple.

Lorsqu'un pilote est chargé en mémoire, il indique au noyau le type de périphérique pour lequel il est compétent, en lui transmettant une structure `file_operations` décrite dans le fichier d'en-tête `<linux/fs.h>`. Ce mécanisme est interne au noyau, mais il est intéressant de le comprendre pour bien saisir le rôle des fichiers spéciaux. À l'instar des méthodes définies pour les classes en programmation orientée objet, la structure `file_operations` contient des pointeurs sur les fonctions que le pilote est capable de fournir pour le périphérique :

Nom du champ	Rôle
lseek	Fonction appelée pour déplacer la position de lecture ou d'écriture sur le périphérique.
read	Fonction de lecture depuis le périphérique.
write	Fonction d'écriture sur le périphérique.
readdir	Fonction de lecture du contenu d'un répertoire servant à homogénéiser l'implémentation des systèmes de fichiers, mais non utilisée sur les périphériques.
poll	Fonction permettant de surveiller la disponibilité des données en lecture ou en écriture ; ceci sera étudié dans le chapitre 30.
ioctl	Point d'entrée permettant d'assurer des opérations particulières sur un périphérique autres que les lectures ou écritures, par exemple éjection d'un CD, programmation de la parité d'une interface série, etc.
mmap	Fonction demandant la projection du contenu du périphérique en mémoire.
open	Fonction d'ouverture et d'initialisation du périphérique.
flush	Demande de vidage des buffers associés à un périphérique.
release	Fonction de fermeture et libération d'un périphérique, équivalent de `close()`.
fsync	Fonction de synchronisation du contenu du périphérique et de ses buffers associés.
fasync	Demande de fonctionnement asynchrone du périphérique.
check_media_change	Fonction vérifiant si le support amovible contenu dans le périphérique a été modifié (par exemple un CD).
revalidate	Fonction de gestion du buffer cache.
lock	Fonction de verrouillage du périphérique.

Lorsqu'on demande l'ouverture d'un fichier spécial de périphérique, par exemple un port série, le noyau vérifie le numéro majeur et appelle la fonction `open()` du pilote correspondant, en lui transmettant diverses informations, dont le numéro mineur désiré. Bien sûr, certains

pilotes de périphériques n'implémentent pas toutes les fonctions indiquées précédemment (on ne peut pas déplacer avec `lseek()` la position de lecture sur un port de communication série), aussi existe-t-il des routines par défaut, permettant de renvoyer une erreur par exemple.

Les numéros majeur et mineur d'un fichier spécial sont donc essentiels pour la compréhension entre le noyau et le pilote de périphérique, contrairement au nom du fichier spécial qui n'a aucune importance. Les numéros réservés sont décrits dans le fichier `Documentation/devices.txt` accompagnant les sources du noyau. Par exemple, les ports de communication série sont gérés avec le numéro majeur 5 et les numéros mineurs 64, 65, 66 pour les ports COM1, COM2, COM3 (nommés généralement `ttyS0`, `ttyS1`, `ttyS2` sous Linux). On peut le constater en examinant les cinquième et sixième colonnes de la commande `ls -l` sur les fichiers spéciaux correspondants :

```
$ cd /dev
$ ls -l cua*
crw-rw-rw-   1 root     root       5,  64 May  5  1998 cua0
crw-------   1 root     root       5,  65 May  5  1998 cua1
crw-------   1 root     root       5,  66 May  5  1998 cua2
crw-------   1 root     root       5,  67 May  5  1998 cua3
$
```

Le caractère '`c`' en tête de la première colonne indique qu'il s'agit d'un périphérique spécial de type caractère. D'autres peuvent être de type bloc :

```
$ ls -l hda1*
brw-rw----   1 root     disk       3,   1 May  5  1998 hda1
brw-rw----   1 root     disk       3,  10 May  5  1998 hda10
brw-rw----   1 root     disk       3,  11 May  5  1998 hda11
brw-rw----   1 root     disk       3,  12 May  5  1998 hda12
brw-rw----   1 root     disk       3,  13 May  5  1998 hda13
brw-rw----   1 root     disk       3,  14 May  5  1998 hda14
brw-rw----   1 root     disk       3,  15 May  5  1998 hda15
brw-rw----   1 root     disk       3,  16 May  5  1998 hda16
$
```

La différence entre périphériques caractère et bloc réside dans la manière d'accéder aux données. Dans un cas, elles arrivent octet par octet, dans l'autre des blocs complets sont affectés pour la lecture ou l'écriture. Un corollaire de cette distinction est qu'un périphérique de type bloc peut généralement contenir un système de fichiers, ce qui n'est pas possible avec un périphérique en mode caractère. L'essentiel des périphériques de type bloc est constitué de disques durs, de lecteurs de disquettes et de CD-Rom. Lorsqu'on développe un driver personnalisé pour assurer l'interface avec un périphérique qui est aussi personnalisé (par exemple un instrument de mesure), on utilise donc généralement un pilote de type caractère.

Pour créer un nouveau nœud du système de fichiers qui représente un fichier spécial de périphérique, on utilise généralement l'utilitaire `/bin/mknod`, qui prend en arguments la lettre `b` ou `c` (suivant le type de périphérique), le numéro majeur et le numéro mineur. Cette application sert ainsi de frontal à l'appel-système `mknod()`, dont le troisième argument regroupe les deux numéros mineurs sous forme d'une valeur `dev_t`, composée ainsi :

```
dev_t periph;
periph = (majeur << 8) | mineur;
```

Le numéro mineur est donc limité à l'espace allant de 0 à 255.

La création directe d'un fichier spécial, en employant l'appel-système `mknod()`, est assez rare, puisqu'on préfère en général créer ces nœuds en utilisant `/bin/mknod`, éventuellement dans un script shell qui encadre le chargement du module pilote du périphérique correspondant.

L'appel-système `mknod()` donnant un accès direct aux périphériques matériels reliés à l'ordinateur, il est naturellement réservé à l'administrateur. De même, les fichiers spéciaux de périphériques ne sont interprétés en tant que tels par le noyau que si la partition a été montée avec l'option adéquate. C'est le cas par défaut pour les systèmes de fichiers montés automatiquement au démarrage, mais ceci est désactivé pour les supports susceptibles d'être montés par n'importe quel utilisateur, comme les disquettes ou les CD-Rom.

Masque de création de fichier

Lorsqu'un processus crée un fichier, quel que soit son type, les permissions d'accès sont filtrées par un masque particulier, qui retire des autorisations. Ce masque peut être modifié avec l'appel-système **`umask()`**, déclaré dans `<sys/stat.h>` :

```
int umask (int masque);
```

Cet appel-système permet de configurer le nouveau masque et renvoie sa valeur précédente. Lorsqu'un processus dispose par exemple d'un masque 0022 en octal, ce qui est courant, même s'il crée un fichier avec les autorisations 0777 (lecture, écriture, exécution pour tout le monde), les bits d'écriture (correspondant à 2) seront supprimés pour le groupe et le reste des utilisateurs. Le fichier ainsi créé aura l'autorisation 0755, ce qui est plus raisonnable. En voici un exemple très simple :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>

    int
main (void)
{
    int  fd;
    int  masque;

    masque = umask(0);
    fprintf(stdout, "Ancien masque = %o, nouveau = 0 \n", masque);
    fprintf(stdout, "Tentative de création de essai.umask \n");
    fd = open("essai.umask", O_RDWR | O_CREAT | O_EXCL, 0777);
    if (fd < 0)
        perror("open");
    else
        close(fd);
    system("ls -l essai.umask");
    unlink("essai.umask");

    umask(masque);
    fprintf(stdout, "Remise masque = %o \n", masque);
    fprintf(stdout, "Tentative de création de essai.umask \n");
```

```
    fd = open("essai.umask", O_RDWR | O_CREAT | O_EXCL, 0777);
    if (fd < 0)
        perror("open");
    else
        close(fd);
    system("ls -l essai.umask");
    unlink("essai.umask");
    return EXIT_SUCCESS;
}
```

L'exécution montre bien la différence des modes lors de la création d'un fichier :

```
$ ./exemple_umask
Ancien masque = 22, nouveau = 0
Tentative de création de essai.umask
-rwxrwxrwx   1 ccb      ccb       0 Jan  3 16:55 essai.umask
Remise masque = 22
Tentative de création de essai.umask
-rwxr-xr-x   1 ccb      ccb       0 Jan  3 16:55 essai.umask
$
```

Le masque étant hérité au cours d'un `fork()` et conservé au cours d'un `exec()`, il s'agit avant tout d'un dispositif de sécurité qu'on peut mettre en place dans le script de configuration du shell (qui dispose d'une commande de configuration `umask` interne) pour s'assurer de la confidentialité des fichiers créés ultérieurement.

Conclusion

Ce chapitre nous a permis de faire le point sur les fichiers en tant qu'entités physiquement présentes sur les disques. De par les différences existant entre les divers types de systèmes de fichiers, nous sommes resté à un niveau de description assez élevé, correspondant à celui d'un développement applicatif.

Pour les lecteurs désireux d'approfondir les concepts sous-jacents au stockage des données sur disque, on conseillera [CARD 1997] *Programmation Linux2.0* par exemple, dans lequel le système de fichiers ext2 est précisément décrit par l'un de ses concepteurs.

Pour des informations sur les versions plus récentes de Linux, je conseille l'excellent [LOVE 2003] *Linux Kernel Development* qui contient également des renseignements sur la plupart des autres sous-ensembles du noyau.

22

Bases de données

Le langage C en général et la bibliothèque GlibC ne sont certainement pas les meilleures solutions pour manipuler des bases de données. Il existe de multiples bibliothèques ou systèmes complets de gestion de bases de données permettant d'obtenir sous Linux des résultats optimaux, tant dans le domaine du logiciel libre comme *PostgreSQL* que commercial.

On n'emploiera donc pas les fonctionnalités présentées dans ce chapitre pour mettre au point de grosses bases de données relationnelles. Néanmoins, il arrive qu'un logiciel ait besoin de gérer une *petite* base de données, comme fonctionnalité annexe par rapport au but principal de l'application.

On peut en trouver un bon exemple en observant un logiciel de composition et d'émission de fax. Les fonctionnalités essentielles concerneront la mise en page et la présentation graphique du message, l'importation des fichiers Postscript issus du traitement de texte, et la gestion du protocole de transfert. Toutefois la présence d'une petite base de données contenant les destinataires habituels avec leurs numéros de téléphone est un atout non négligeable de notre logiciel.

De plus en plus d'applications permettent de transmettre directement des résultats, des messages, ou des rapports de bogues par courrier électronique, et peuvent aussi tirer profit de manière sensible d'un annuaire des correspondants. Un degré de convivialité supplémentaire est encore atteint si plusieurs applications indépendantes utilisent le même annuaire.

Nous étudierons en premier lieu les bases de données DBM, qui sont un héritage historique des premiers Unix, ainsi que leurs extensions NDBM. Ensuite, nous examinerons toute l'interface Gnu pour les bases de données GDBM.

Une seconde classe de bases de données, nommées DB Berkeley, est disponible avec la GlibC. Plus performants que les précédents, ces mécanismes seront étudiés par la suite dans ce chapitre.

Pour l'ensemble des exemples décrits ci-dessous, nous aurons besoin de disposer d'un minimum de données consistantes. Pour cela nous utiliserons le fichier `/usr/src/linux/CREDITS`,

qui est présent sur toutes les stations où les sources du noyau sont installées. Ce fichier est formaté de cette façon :

- Un en-tête écrit par Linus Torvalds présente les données se trouvant à la suite. Cet en-tête se termine par une chaîne de tirets « `----------` » isolée sur une ligne.
- Des blocs de données séparés par une ligne vierge décrivent les informations concernant les développeurs du noyau.
- Au sein de chaque bloc, les trois premiers caractères de chaque ligne identifient le champ correspondant (le troisième caractère est un espace) ainsi :

Symbole	Contenu
`D:`	Description du travail accompli par le contributeur
`E:`	Adresse e-mail
`N:`	Prénoms et nom
`P:`	Clé de validation PGP
`S:`	Adresse postale
`W:`	Adresse web

Des lignes de commentaires peuvent apparaître, précédées du symbole dièse #.

Voici un extrait du fichier contenu dans les sources du noyau 2.6.9 :

```
        This is at least a partial credits-file of people that have
        contributed to the Linux project.  It is sorted by name and
        formatted to allow easy grepping and beautification by
        scripts.  The fields are: name (N), email (E), web-address
        (W), PGP key ID and fingerprint (P), description (D), and
        snail-mail address (S).
        Thanks,

                        Linus
----------

N: Matti Aarnio
E: mea@nic.funet.fi
D: Alpha systems hacking, IPv6 and other network related stuff
D: One of assisting postmasters for vger.rutgers.edu's lists
S: (ask for current address)
S: Finland

N: Dragos Acostachioaie
E: dragos@iname.com
W: http://www.arbornet.org/~dragos
D: /proc/sysvipc
S: C. Negri 6, bl. D3
S: Iasi 6600
S: Romania

[...]
```

```
N: Marc Zyngier
E: maz@wild-wind.fr.eu.org
D: MD driver
S: 11 rue Victor HUGO
S: 95560 Montsoult
S: France

# Don't add your name here, unless you really _are_ after Marc
# alphabetically. Leonard used to be very proud of being the
# last entry, and he'll get positively pissed if he can't even
# be second-to-last.  (and this file really _is_ supposed to be
# in alphabetic order)
```

Nous allons donc écrire une routine capable de parcourir ce fichier et de remplir des chaînes de caractères contenant les champs *nom* (et prénoms), *adresse e-mail* et *site web*. Comme certains champs peuvent s'étendre sur plusieurs lignes du fichier original, on invoquera `realloc()` pour allonger les chaînes comme il le faut.

À la fin de chaque bloc, notre routine fera appel à une fonction d'ajout dans la base de données, qui variera suivant les interfaces utilisées.

Bases de données Unix DBM

Les premières bases de données qui furent largement répandues sous Unix étaient gérées par une interface nommée DBM pour *Database Manager*. Peu performantes, ces routines ne permettaient pas de manipuler simultanément plus d'une base par programme ni de gérer correctement un accès concurrentiel.

Une interface améliorée fut définie par la suite et nommée NDBM pour *New DBM*. Elle autorisait un verrouillage du fichier pour éviter les accès simultanés et ajoutait la possibilité de manipuler plusieurs bases dans la même application.

Finalement, le projet Gnu fournit une bibliothèque nommée GDBM, offrant quelques améliorations et une compatibilité ascendante avec DBM et NDBM. Nous étudierons ces trois interfaces successivement.

Pour utiliser la bibliothèque GDBM, il faut ajouter l'argument `–lgdbm` en ligne de commande de l'éditeur de liens. Les fichiers d'en-tête à inclure sont les suivants :

Interface	Fichier d'en-tête
DBM	`<gdbm/dbm.h>`
NDBM	`<gdbm/ndbm.h>`
GDBM	`<gdbm.h>` ou `<gdbm/gdbm.h>`

Les bases DBM permettent de stocker des associations entre des données et des clés. Le stockage est assuré par l'intermédiaire d'une table de hachage extensible, comme nous en avons déjà examinée dans le chapitre 17. La base de données est enregistrée dans deux fichiers complémentaires, avec les suffixes `.pag` et `.dir`. Sous Linux, un seul fichier est utilisé. Pour conserver toutefois l'aspect original de l'interface DBM historique, avec ses deux fichiers, la bibliothèque crée deux liens matériels sur le même contenu.

Les données, comme les clés, sont représentées par un type `datum`, structure contenant au minimum les champs suivants :

Nom	Type	Signification
`dptr`	`char *`	Donnée proprement dite
`dsize`	`int`	Longueur des données

Pour ouvrir une base de données, on emploie la fonction **`dbminit()`**. Elle prend en argument le nom de la base (sans les extensions `.pag` et `.dir`). Ces deux fichiers doivent exister avant d'appeler `dbminit()`.

```
int dbminit (const char * nom);
```

Si les fichiers n'existent pas, `dbminit()` les crée avec un mode d'accès entièrement nul, ne permettant donc aucune manipulation. On pourra donc employer systématiquement avant `dbminit()` un code du genre :

```
    if ((fp = fopen ("base_de_donnees.pag", "a")) != NULL)
        fclose (fp);
    if ((fp = fopen ("base_de_donnees.dir", "a")) != NULL)
        fclose (fp);
    dbminit ("base_de_donnees");
```

La fonction `dbminit()` de la bibliothèque GDBM rétablira correctement les deux liens matériels sur le même fichier.

Si `dbminit()` échoue, elle renvoie –1 au lieu de 0 et remplit la variable `errno`. Avec la bibliothèque GDBM, la base est verrouillée durant l'ouverture – avec un verrou consultatif –, ne permettant donc qu'un seul accès à la fois.

La fermeture de la base se fait avec la fonction **`dbmclose()`** :

```
int dbmclose (void);
```

Pour enregistrer une paire clé/donnée dans la base, on emploie la fonction **`store()`**. Son prototype est le suivant :

```
int store (datum cle, datum donnee);
```

Si `store()` réussit, elle renvoie zéro. Si une erreur se produit, elle renvoie –1 et remplit `errno`. Si la clé existe déjà dans la base, cette fonction transmet une valeur positive. La clé doit donc être un identifiant unique. Faute d'en avoir dans la liste des contributeurs de Linux, nous allons en créer une artificiellement en numérotant les enregistrements. Cette méthode n'est pas très intelligente car nous ne stockons pas le nombre de données dans la base. Si on veut ajouter ultérieurement d'autres contributeurs, il faudra balayer toute la base afin de compter les enregistrements pour créer une nouvelle clé.

Pour enregistrer nos données Nom et prénoms, Adresse e-mail et Site web en une seule fois, nous mettons les chaînes bout à bout. Le programme suivant permet de créer une base de données DBM en lisant le fichier des contributeurs :

cree_dbm.c :

```
    #define _GNU_SOURCE /* pour stpcpy() */
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <gdbm/dbm.h>

    static void construit_base (void);

    static char * fichier_credits = "CREDITS";

    int
main (int argc, char * argv[])
{
    FILE * fp;
    char * fichier;
    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_base \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fichier = malloc(strlen(argv[1]) + 5);
    strcpy(fichier,argv[1]);
    strcat(fichier,".pag");
    if ((fp = fopen(fichier, "a")) != NULL)
        fclose(fp);
    strcpy(fichier, argv[1]);
    strcat(fichier, ".dir");
    if ((fp = fopen(fichier, "a")) != NULL)
        fclose(fp);
    free(fichier);
    if (dbminit(argv[1]) != 0) {
        perror("dbminit");
        exit(EXIT_FAILURE);
    }
    construit_base();
    dbmclose();
    return EXIT_SUCCESS;
}

    static void
construit_base (void)
{
    FILE * fichier;
    char   ligne[256];
    char * fin_ligne;
    size_t debut_ligne;
    int    i = 0;
    char * nom   = NULL;
    char * email = NULL;
    char * web   = NULL;
    char * chaine;
    datum  cle;
```

```
datum  donnee;
int    retour;

if ((fichier = fopen(fichier_credits, "r")) == NULL) {
    perror(fichier_credits);
    return;
}
/* Sauter l'en-tete */
while (1) {
    if (fgets(ligne, 256, fichier) == NULL)
        return;
    /* Supprimer commentaires et retours chariot */
    if ((fin_ligne = strpbrk(ligne, "\n\r#")) != NULL)
        fin_ligne[0] = '\0';
    if (strncmp(ligne, "--", 2) == 0)
        break;
}
while (1) {
    if (fgets(ligne, 256, fichier) == NULL)
        return;
    if ((fin_ligne = strpbrk(ligne, "\n\r#")) != NULL)
        fin_ligne[0] = '\0';
    /* Supprimer blancs en début de ligne */
    if ((debut_ligne = strspn(ligne, " \t\n\r")) != 0)
        memmove(ligne, ligne + debut_ligne,
                 strlen(ligne + debut_ligne) + 1);
    if (strlen(ligne) == 0) {
        /* Ligne vide. Si le bloc est prêt, on le stocke */
        if (nom != NULL) {
            cle.dptr = (char *) (& i);
            cle.dsize = sizeof(int);
            /* On colle les champs bout à bout */
            donnee.dsize = 0;
            if (nom != NULL)
                donnee.dsize += strlen(nom);
            donnee.dsize ++; /* caractère nul */
            if (email != NULL)
                donnee.dsize += strlen(email);
            donnee.dsize ++;
            if (web != NULL)
                donnee.dsize += strlen(web);
            donnee.dsize ++;
            donnee.dptr = malloc(donnee . dsize);
            if (donnee.dptr != NULL) {
                memset(donnee.dptr, '\0', donnee.dsize);
                chaine = donnee.dptr;
                if (nom != NULL)
                    chaine = stpcpy(chaine, nom);
                chaine ++; /* caractère nul */
                if (email != NULL)
                    chaine = stpcpy(chaine, email);
                chaine ++;
```

```
                if (web != NULL)
                    chaine = stpcpy(chaine, web);
                /*
                 * ENREGISTREMENT DES DONNÉES
                 */
                retour = store(cle, donnee);
                if (retour < 0)
                    perror("store");
                if (retour > 0)
                    fprintf(stderr, "Doublon\n");
                free(donnee.dptr);
                donnee.dptr = NULL;
            }
            i ++;
        }
        /* On libère les chaînes allouées */
        if (nom != NULL)     free(nom);
        if (email != NULL)   free(email);
        if (web != NULL)     free(web);
        nom   = NULL;
        email = NULL;
        web   = NULL;
        continue;
    }
    if (strncmp(ligne, "N: ", 3) == 0) {
        if (nom == NULL) {
            if ((nom = malloc(strlen(ligne) - 2)) != NULL)
                strcpy(nom, & (ligne[3]));
            continue;
        }
        chaine = realloc(nom, strlen(nom) + strlen(ligne) - 1);
        if (chaine == NULL)
            continue;
        nom = chaine;
        sprintf(nom, "%s %s", nom, & (ligne[3]));
        continue;
    }
    if (strncmp(ligne, "E: ", 3) == 0) {
        if (email == NULL) {
            if ((email = malloc(strlen(ligne) - 2)) != NULL)
                strcpy(email, & (ligne[3]));
            continue;
        }
        chaine = realloc(email, strlen(email) + strlen(ligne) - 1);
        if (chaine == NULL)
            continue;
        email = chaine;
        sprintf(email, "%s %s", email, & (ligne[3]));
        continue;
    }
    if (strncmp(ligne, "W: ", 3) == 0) {
        if (web == NULL) {
```

```
                if ((web = malloc(strlen(ligne) - 2)) != NULL)
                    strcpy(web, & (ligne[3]));
                continue;
            }
            chaine = realloc(web, strlen(web) + strlen(ligne) - 1);
            if (chaine == NULL)
                continue;
            web = chaine;
            sprintf(web, "%s %s", web, & (ligne[3]));
            continue;
        }
    }
    fclose(fichier);
}
```

Commençons donc par créer notre base :

```
$ ./cree_dbm credits
$ ls -l credits*
-rw-rw-r--   2 ccb      ccb     34892 Feb 15 17:37 credits.dir
-rw-rw-r--   2 ccb      ccb     34892 Feb 15 17:37 credits.pag
$
```

Les deux fichiers sont en réalité deux liens physiques sur le même fichier (on le remarque grâce au 2 en seconde colonne).

Pour lire les informations contenues dans une base, on emploie la fonction **`fetch()`**, qui renvoie la donnée associée à une clé :

```
datum fetch (datum cle);
```

Si la clé n'existe pas dans la base, le champ `dptr` de la structure `datum` renvoyée est `NULL`. Nous créons donc un programme qui recherche les enregistrements associés aux numéros passés sur la ligne de commande à la suite du nom de la base :

cherche_cle_dbm.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <gdbm/dbm.h>

    void affiche_contributeur (datum cle, datum donnee);

    int
main (int argc, char * argv[])
{
    datum  cle;
    datum  donnee;
    int    i;
    int    numero;

    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s nom_base clés...\n", argv[0]);
        exit(EXIT_FAILURE);
```

```
    }
    if (dbminit(argv[1]) != 0) {
        perror("dbminit");
        exit(EXIT_FAILURE);
    }
    for (i = 2; i < argc; i ++) {
        if (sscanf(argv[i], "%d", & numero) == 1) {
            cle.dptr = (char *) (& numero);
            cle.dsize = sizeof(int);
            donnee = fetch(cle);
            if (donnee.dptr != NULL)
                affiche_contributeur(cle, donnee);
            else
                fprintf(stderr, "%s : inconnu\n", argv[i]);
        }
    }
    dbmclose();
    return EXIT_SUCCESS;
}

    void
affiche_contributeur (datum cle, datum donnee)
{
    char * nom;
    char * email;
    char * web;

    nom = donnee.dptr;
    email = & (nom[strlen(nom)   + 1]);
    web   = & (email[strlen(email) + 1]);
    fprintf(stdout, "Numero : %d\n", * ((int *) cle . dptr));
    fprintf(stdout, "  Nom      : %s\n", nom);
    fprintf(stdout, "  Email    : %s\n", email);
    fprintf(stdout, "  Web      : %s\n", web);
}
```

Nous pouvons déjà interroger notre base ainsi :

```
$ ./cherche_cle_dbm credits 31 411 500
Numero : 31
  Nom      : Donald Becker
  Email    : becker@cesdis.gsfc.nasa.gov
  Web      :
Numero : 411
  Nom      : Theodore Ts'o
  Email    : tytso@mit.edu
  Web      :
500 : inconnu
$
```

Les données renvoyées par `fetch()` doivent être considérées comme se trouvant dans une zone de données statique, susceptible d'être écrasée à chaque appel.

Si on désire effacer un enregistrement, il suffit d'utiliser la fonction **delete()**, qui renvoie 0 si elle réussit ou une valeur négative si l'enregistrement n'est pas trouvé. Lorsqu'un enregistrement est effacé, le fichier n'est pas réduit pour autant. Par contre, l'espace sera réutilisé par la suite.

```
int delete (datum cle);
```

Il est possible de balayer séquentiellement le fichier au moyen des routines **first_key()** et **next_key(),** qui renvoient respectivement la première clé du fichier et la clé se trouvant après celle qui est passée en argument.

```
datum first_key (void);
datum next_key (datum cle);
```

Lorsque la fin du fichier est atteinte, le champ `dptr` de la clé renvoyée est `NULL`. On utilise généralement un balayage du genre :

```
for (cle=first_key(); cle . dptr != NULL; cle = next_key (cle)) {
  [...]
}
```

Les bases de données DBM étant organisées sous forme de tables de hachage, l'ordre des éléments renvoyés par `firstkey()` et `nextkey()` est imprévisible, et peut varier lors d'une modification de la base.

Le programme suivant permet de rechercher un nom dans la base, en parcourant tous les enregistrements et en utilisant `strstr()` pour sélectionner les contributeurs à afficher.

cherche_nom_dbm.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <gdbm/dbm.h>

    void affiche_contributeur (datum cle, datum donnee);

    int
main (int argc, char * argv[])
{
    datum  cle;
    datum  donnee;
    char   chaine[256];
    char * fin_chaine;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_base\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (dbminit(argv[1]) != 0) {
        perror("dbminit");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stdout, "(Nom)> ");
```

```
        if (fgets(chaine, 256, stdin) == NULL)
            break;
        if ((fin_chaine = strpbrk(chaine, "\n\r ")) != NULL)
            * fin_chaine = '\0';
        if (strlen(chaine) == 0)
            continue;
        for (cle = firstkey(); cle.dptr != NULL; cle = nextkey(cle)) {
            donnee = fetch(cle);
            if (donnee.dptr != NULL)
                if (strstr(donnee.dptr, chaine) != NULL)
                    affiche_contributeur(cle, donnee);
        }
    }
    fprintf(stdout, "\n");
    dbmclose();
    return EXIT_SUCCESS;
}
```

Nous pouvons interroger à nouveau la base :

```
$ ./cherche_nom_dbm credits
(Nom)> Linus
Numero : 404
  Nom      : Linus Torvalds
  Email    : torvalds@odsl.org
  Web      :
(Nom)> Alan
Numero : 85
  Nom      : Alan Cox
  Email    :
  Web      : http://www.linux.org.uk/diary/
(Nom)> Andrea
Numero : 196
  Nom      : Andrea Jaeger
  Email    : aj@suse.de
  Web      :
Numero : 12
  Nom      : Andrea Arcangeli
  Email    : andrea@suse.de
  Web      : http://www.kernel.org/pub/linux/kernel/people/andrea
[...]
(Nom)> (Contrôle-D)
$
```

Par la même occasion, nous remarquons que la bibliothèque GDBM verrouille l'accès à la base de données, même si ce comportement n'est pas celui de l'interface DBM originale. Pour cela, on lance `cherche_nom_dbm` simultanément sur deux terminaux, et la seconde invocation échoue ainsi :

```
$ ./cherche_nom_dbm credits
dbminit: Ressource temporairement non disponible
$
```

Les fichiers DBM créés par la bibliothèque GDBM ne sont pas compatibles avec les fichiers DBM Unix traditionnels. Si on désire assurer une conversion, on peut utiliser le programme `conv2gdbm` qui est fourni avec cette bibliothèque. Par contre, les interfaces DBM et NDBM que nous allons examiner sont en réalité émulées à partir de l'interface GDBM, qui est plus complète. Les bases de données sont donc totalement compatibles quelle que soit l'interface choisie. Enfin, contrairement aux implémentations Unix traditionnelles, les bases de données DBM ne contiennent pas de trous, comme nous en avions observé dans le chapitre 18, et leurs tailles n'augmentent pas si on les copie avec les utilitaires classiques `cp`, `tar`…

Bases de données Unix NDBM

L'interface DBM présente une déficience évidente avec l'impossibilité de manipuler plusieurs bases de données en même temps et l'interdiction d'utiliser la même base dans plusieurs processus concurrents. Elle a donc été étendue par l'interface NDBM, qui corrige ces deux points.

Pour pouvoir gérer plusieurs bases de données simultanément, on introduit un type opaque **`DBM`**, qu'on peut comparer au type `FILE`.

La fonction d'ouverture **`dbm_open()`** est également un peu plus complexe puisqu'elle inclut deux arguments supplémentaires :

```
DBM * dbm_open (const char * nom, int attributs, int mode);
```

Ces deux derniers arguments sont identiques à ceux de l'appel-système `open()`. Ils permettent d'ouvrir une base de données en lecture seule (`O_RDONLY`), ou en lecture et écriture (`O_RDWR`). Si on tente d'ouvrir la base en écriture seule, la bibliothèque transforme automatiquement l'attribut en lecture et écriture. Si on crée une nouvelle base (avec `O_CREAT`), le dernier argument permet de préciser les autorisations à donner aux fichiers.

Il est à présent possible non seulement de manipuler plusieurs bases de données simultanément dans le même programme, mais aussi d'ouvrir la même base dans plusieurs processus différents en même temps si l'accès se fait partout en lecture seule. Le verrouillage géré par la bibliothèque permet en effet de disposer de plusieurs processus lecteurs en parallèle. Bien sûr, si un processus obtient une ouverture en lecture et écriture, aucun autre accès n'est possible en même temps – pas même en lecture seule.

Si `dbm_open()` échoue, elle renvoie un pointeur `NULL` et remplit `errno`.

Toutes les fonctions de l'interface DBM se retrouvent, avec NDBM, gratifiées d'un premier argument supplémentaire représentant la base de données et d'un préfixe permettant de les distinguer. On trouve ainsi **`dbm_close()`**, **`dbm_fetch()`**, **`dbm_delete()`**, ainsi que **`dbm_firstkey()`** et **`dbm_nextkey()`**, dont les prototypes sont les suivants :

```
void  dbm_close    (DBM * fichier);
datum dbm_fetch    (DBM * fichier, datum cle);
int   dbm_delete   (DBM * fichier, datum cle);
datum dbm_firstkey (DBM * fichier);
datum dbm_nextkey  (DBM * fichier, datum cle);
```

La fonction **`dbm_store()`** dispose encore d'un argument supplémentaire :

```
int dbm_store (DBM * fichier, datum cle, datum donnee, int attribut);
```

L'attribut indiqué en dernière position peut prendre l'une des deux valeurs suivantes :

Nom	Signification
DBM_INSERT	On ajoute l'élément dans la base, à condition que la clé ne s'y trouve pas déjà. Sinon la fonction échoue sans modifier quoi que ce soit.
DBM_REPLACE	On ajoute l'élément dans la base si sa clé ne s'y trouve pas. Si un autre élément possède déjà la même clé, il est remplacé. Naturellement, il n'y a toujours qu'une seule occurrence d'une clé donnée.

Quelques routines ont été ajoutées par rapport à l'interface DBM. Les fonctions **dbm_error()** et **dbm_clearerr()** par exemple permettent de consulter ou d'effacer l'indicateur d'erreur de la base de données. Ces fonctions ressemblent à `ferror()` et à `clearerr()` que nous avions rencontrées dans le chapitre 18.

```
int dbm_error (DBM * fichier);
int dbm_clearerr (DBM * fichier);
```

Les fonctions **dbm_pagfno()** et **dbm_dirfno()** doivent renvoyer le numéro des descripteurs de fichiers correspondant aux fichiers `.pag` et `.dir`. Naturellement, ces deux fichiers étant identiques dans l'implémentation GDBM, il n'y a qu'un seul descripteur utilisé.

```
int dbm_pagfno (DBM * fichier);
int dbm_dirfno (DBM * fichier);
```

La fonction **dbm_rdonly()** renvoie une valeur booléenne indiquant si la base de données a été ouverte en lecture seule.

```
int dbm_rdonly (DBM * fichier);
```

Pour manipuler les fonctions de l'interface NDBM, on peut très bien utiliser la base de données que nous avons constituée dans le paragraphe précédent. Il suffit en pratique d'ajouter le préfixe `dbm_` aux fonctions employées et de gérer l'argument `DBM *` supplémentaire. Nous pouvons par exemple modifier le programme `cherche_nom_dbm.c` ainsi :

cherche_nom_nbdm.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <gdbm/ndbm.h>

    void affiche_contributeur (datum cle, datum donnee);

    int
main (int argc, char * argv[])
{
    datum  cle;
    datum  donnee;
    DBM *  dbm;
    char   chaine[256];
    char * fin_chaine;
```

```
    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_base\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((dbm = dbm_open(argv[1], O_RDONLY, 0)) == NULL) {
        perror("dbm_open");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stdout, "(Nom)> ");
        if (fgets(chaine, 256, stdin) == NULL)
            break;
        if ((fin_chaine = strpbrk(chaine, "\n\r ")) != NULL)
            * fin_chaine = '\0';
        if (strlen(chaine) == 0)
            continue;
        for (cle = dbm_firstkey(dbm);
             cle.dptr != NULL;
             cle = dbm_nextkey(dbm, cle)) {
            donnee = dbm_fetch(dbm, cle);
            if (donnee.dptr != NULL)
                if (strstr(donnee.dptr, chaine) != NULL)
                    affiche_contributeur(cle, donnee);
        }
    }

    fprintf(stdout, "\n");
    dbm_close(dbm);
    return EXIT_SUCCESS;
}
```

On peut alors lancer plusieurs sessions de cette application dans différentes fenêtres X-Term et observer que l'accès simultané est possible si les processus ouvrent la base de données en lecture seule.

Bases de données Gnu GDBM

L'interface NDBM, sans être très performante en termes de fonctionnalités de gestion de bases de données, est quand même assez largement utilisée sous Unix, par exemple pour stocker les informations concernant les services réseau avec NIS. La bibliothèque GDBM ajoute quelques extensions Gnu à cette interface.

Tout d'abord, on notera que les fonctions sont à présent préfixées par `gdbm_` et que le type représentant une base de données ouverte est **`GDBM_FILE`**.

> **Attention**
> Le type `GDBM_FILE` étant déjà un pointeur, on le manipule directement et pas sous la forme `GDBM_FILE *`.

La routine **gdbm_open()** d'ouverture d'une base de données est légèrement étendue :

```
GDBM_FILE gdbm_open (const char * nom,
                     int taille_bloc,
                     int attributs,
                     int mode,
                     void (* fonction_erreur) (const char * message));
```

Le premier argument est le nom de la base de données. Contrairement aux fonctions d'ouverture des interfaces DBM et NDBM, il s'agit ici du nom complet du fichier. Si on désire accéder à la base construite précédemment, on transmettra donc le nom du fichier `credits.pag` ou `credits.dir`.

Le second argument n'est utilisé que lors de la création de la base. Il s'agit de la taille des blocs employés pour la lecture ou l'écriture des données. Cette valeur doit être supérieure ou égale à 512 pour être prise en compte, sinon la bibliothèque se sert de la fonction `fstat()` pour déterminer une taille de bloc optimale. En général, on prendra donc une valeur nulle.

Le troisième argument est un attribut qui doit d'abord comporter l'une des constantes suivantes :

Nom	Signification
`GDBM_READER`	Ouverture d'une base de données existante en lecture seule. Plusieurs accès de ce type sont possibles de manière concurrente.
`GDBM_WRITER`	Ouverture d'une base de données existante en lecture et écriture. Un seul processus peut avoir accès à la base.
`GDBM_WRCREAT`	Ouverture d'une base de données en lecture et écriture. Si la base n'existe pas, elle est créée.
`GDBM_NEWDB`	Ouverture d'une base de données en lecture et écriture. Si la base n'existe pas, elle est créée. Si elle existe déjà, elle est écrasée.

De plus, ce troisième argument peut également comprendre, par un OU binaire, les constantes suivantes :

Nom	Signification
`GDBM_SYNC`	Synchronisation des écritures. Les modifications sur la base sont transmises immédiatement au contrôleur de disque. Les performances sont légèrement dégradées.
`GDBM_NOLOCK`	Pas de gestion du verrouillage de la base. L'application doit fournir sa propre méthode pour éviter les problèmes d'accès simultanés.

Le quatrième argument, le mode, correspond aux permissions d'accès qui seront installées sur les fichiers de la base de données s'ils sont créés. On emploie souvent 0644 ou 0640.

Finalement, le dernier argument de `gdbm_open()` est un pointeur sur une fonction d'erreur, qui sera invoquée en cas de détection d'un problème fatal sur la base. Cette fonction prend en argument une chaîne de caractères correspondant au message d'erreur indiquant le problème. Si on transmet un pointeur `NULL`, la bibliothèque GDBM fournit un gestionnaire d'erreur standard.

La fonction `gdbm_open()` renvoie un objet de type `GDBM_FILE`, ou `NULL` en cas d'erreur. La valeur d'erreur est transmise dans une variable globale nommée **gdbm_errno**. Celle-ci est de

type **gdbm_error**. Il existe une fonction permettant d'obtenir un libellé dans une chaîne de caractères statique :

```
char * gdbm_strerror (gdbm_error erreur);
```

Les fonctions suivantes ont un comportement équivalent à celui de la bibliothèque NDBM :

```
void  gdbm_close    (GDBM_FILE fichier);
datum gdbm_fetch    (GDBM_FILE fichier, datum cle);
int   gdbm_delete   (GDBM_FILE fichier, datum cle);
datum gdbm_firstkey (GDBM_FILE fichier);
datum gdbm_nextkey  (GDBM_FILE fichier, datum cle);
int   gdbm_store    (GDBM_FILE fichier, datum cle,
                     datum donnee, int attribut);
```

Le dernier argument de la fonction `gdbm_store()` doit correspondre à l'une des valeurs suivantes :

Nom	Signification
`GDBM_INSERT`	Équivalent de `DBM_INSERT` pour la bibliothèque NDBM
`GDBM_REPLACE`	Équivalent de `DBM_REPLACE` pour la bibliothèque NDBM

La bibliothèque GDBM ajoute également plusieurs routines. La fonction **gdbm_reorganize()** peut être invoquée pour « nettoyer » la base de données lorsqu'il y a eu beaucoup de suppressions successives.

```
int gdbm_reorganize (GDBM_FILE fichier);
```

Cette routine permet de récupérer l'espace libéré sur le disque. Sinon la base de données le conservera, pour le réutiliser par la suite.

Avec la fonction **gdbm_sync()**, on peut demander la synchronisation de la base de données sur le disque, avec son contenu en mémoire. Cette routine n'est pas nécessaire si on l'a ouverte avec l'attribut `GDBM_SYNC`.

```
void gdbm_sync (GDBM_FILE fichier);
```

La fonction **gdbm_exist()** permet de vérifier si une clé est présente dans la base.

```
int gdbm_exist (GDBM_FILE fichier, datum cle);
```

Nous avons vu que la bibliothèque GDBM permet d'ouvrir une base de données sans faire de verrouillage (avec l'option `GDBM_NOLOCK`). L'application doit alors implémenter son propre mécanisme de synchronisation pour l'accès au fichier. Il existe une fonction nommée **gdbm_fdesc()** renvoyant le descripteur associé à la base de données :

```
int gdbm_fdesc (GDBM_FILE fichier);
```

Notons que la bibliothèque GDBM permet de configurer certaines options grâce à une fonction **gdbm_setopt()**, mais qu'il s'agit essentiellement de mécanismes internes de la base, qui sont donc ici hors de notre propos. Pour plus de détails, on pourra se reporter à la documentation de cette bibliothèque.

Afin d'utiliser les routines GDBM, nous allons créer un petit programme permettant de parcourir la base pour afficher tout son contenu :

parcours_gdbm.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <gdbm.h>

    void affiche_contributeur (datum cle, datum donnee);

    int
main (int argc, char * argv[])
{
    GDBM_FILE base;
    datum  cle;
    datum  donnee;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_base \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((base = gdbm_open(argv[1], 0, GDBM_READER, 0, NULL)) == NULL ) {
        fprintf(stderr, "%s : %s\n", argv[1],gdbm_strerror(gdbm_errno));
        exit(EXIT_FAILURE);
    }
    for (cle = gdbm_firstkey(base);
         cle.dptr != NULL;
         cle = gdbm_nextkey(base, cle)) {
        donnee = gdbm_fetch(base, cle);
            if (donnee.dptr != NULL)
                affiche_contributeur(cle, donnee);
    }
    gdbm_close(base);
    return EXIT_SUCCESS;
}
```

Lors de l'exécution de ce programme, nous pouvons vérifier que le nom à transmettre est bien celui du fichier complet et, par la même occasion, faire fonctionner la routine `gdm_strerror()`.

```
$ ./parcours_gdbm credits
credits : File open error
$ ./parcours_gdbm credits.pag
Numero : 136
  Nom      : Bob Frey
  Email    : bobf@advansys.com
  Web      :
Numero : 208
  Nom      : Jan Kara
 [...]
```

```
Numero : 97
  Nom      : Todd J. Derr
  Email    : tjd@fore.com
  Web      : http://www.wordsmith.org/~tjd
Numero : 321
  Nom      : Yuri Per
  Email    : yuri@pts.mipt.ru
  Web      :
$
```

Pour terminer, on peut dire que les bases de données GDBM sont fiables et simples d'utilisation mais que leur champ d'application est assez restreint. Pour obtenir un comportement optimal de la base, il faut que les conditions suivantes soient remplies :

- Accès concurrents limités à la lecture. Plusieurs processus peuvent accéder simultanément à la base de données à condition qu'ils réclament tous un accès en lecture seule. Si un processus demande un accès en écriture, il devra attendre qu'il n'y ait plus de lecteur sur la base. De la même façon, tant qu'un écrivain garde un accès sur la base, il n'y a pas de consultation possible.
- Modifications rares et groupées. En corollaire de la première condition, on devine qu'il est largement préférable que les écritures soient groupées afin d'éviter de mobiliser la base trop fréquemment. En cas de suppression de plusieurs enregistrements, on peut demander une réorganisation des données pour récupérer de la place sur le disque.
- Existence d'une clé unique identifiant les données. La présence d'une clé unique est parfois problématique. D'autant que l'interface DBM ne permet pas de rechercher un enregistrement à l'aide de clés secondaires. Pour organiser une base de données contenant des individus, cela peut poser un véritable problème[1].

La liste des hôtes appartenant à un réseau local est un bon exemple de base de données susceptible d'être gérée en utilisant l'interface DBM. Il existe de fait plusieurs identifiants uniques pouvant servir de clé (adresse IP, adresse MAC, nom complet). La modification de la base est généralement assez rare et peut être considérée comme une opération de maintenance avec interruption du service. Les consultations simultanées d'une base centralisée ont lieu en lecture seule.

Dans ces circonstances, on emploiera avec confiance la bibliothèque GDBM, en profitant en plus de la portabilité de son interface NDBM sur de nombreux systèmes Unix.

Bases de données DB Berkeley

Il existe dans la bibliothèque GlibC une interface permettant de manipuler un second type de bases de données : les DB Berkeley. Ces bases de données peuvent être organisées sous forme de tables de hachage, d'arbres binaires ou d'enregistrements numérotés, au gré de l'utilisateur. Il existe une interface générique pour accéder à toutes les fonctionnalités.

1. Au niveau de l'état civil, du moins en France, l'unicité est garantie à condition de considérer le quadruplet constitué des nom, prénom usuel, date et lieu de naissance. L'utilisation en clé d'accès n'est pas possible car cela ne permet pas la moindre tolérance d'erreur.

Cette bibliothèque est fournie sous Linux par la société *Sleepycat Software* sous licence Open Source. Elle est incluse dans les versions actuelles de la GlibC.

Il y a plusieurs versions de l'interface d'accès aux bases de données DB Berkeley. Nous ne présenterons ici que la plus simple d'entre elles, construite autour de la seule fonction `dbopen()`.

Les autres versions de cette bibliothèque offrent des possibilités très larges, notamment en ce qui concerne les mécanismes transactionnels et les accès concurrents. Mais cela dépasserait le cadre de notre étude. Pour plus de renseignements, le lecteur pourra se reporter à la documentation disponible sur le site web *http://www.sleepycat.com*.

Les bases de données DB Berkeley sont exploitables grâce à une interface en langage C mais également en C++, Java, Perl, Python ou Tcl. Les applications en langage C doivent inclure le fichier d'en-tête `<db_185>`, et il faut ajouter l'option `–ldb` sur la ligne de commande de l'éditeur de liens.

Pour manipuler les clés et les données, on utilise un même type nommé **`DBT`**. Il s'agit en fait d'une structure contenant plusieurs champs, mais nous ne nous servirons que de deux d'entre eux :

Nom	Type	Signification
`data`	`void *`	Pointeur vers la donnée proprement dite
`size`	`size_t`	Longueur de la donnée

Il existe d'autres membres dans les objets `DBT`, aussi faut-il veiller à les initialiser correctement à zéro avant de les employer. On procède ainsi :

```
memset (& dbt, 0, sizeof (DBT));
```

On accède à une base de données en invoquant la fonction **`dbopen()`**, déclarée ainsi :

```
DB * dbopen (const char * nom_fichier, int attributs, int mode
             DBTYPE type, const void * configuration);
```

Les trois premiers arguments de cette routine sont identiques à ceux de l'appel-système `open()`. La plupart du temps, on prendra donc `O_RDWR | O_CREAT` pour l'attribut et 0644 pour le mode. Il est possible de passer un nom de fichier `NULL` si on désire uniquement manipuler la base de données en mémoire, sans la sauvegarder sur le disque.

Le quatrième argument est un type énuméré pouvant prendre l'une des valeurs suivantes :

Nom	Signification
`DB_BTREE`	La base de données est organisée sous forme de structure d'arbre binaire. L'accès aux données est très rapide, mais leur destruction ne permet pas de récupérer l'espace libéré.
`DB_HASH`	La base de données est construite comme une table de hachage extensible.
`DB_RECNO`	La base de données est constituée d'un ensemble d'enregistrements numérotés successifs. Cette structure est surtout intéressante pour stocker des données de tailles constantes.

Finalement, le dernier argument est un pointeur vers une structure de données spécifique au type de base, et permettant de la configurer finement. Si on désire employer une telle structure, il faudra l'indiquer à chaque utilisation ultérieure de la base. Sinon, on peut transmettre

un pointeur `NULL` pour utiliser les paramètres par défaut. Pour avoir plus de précisions sur les paramètres et les possibilités propres à chaque type de base, on pourra se reporter aux pages de manuel `btree(3)`, `hash(3)` et `recno(3)`.

La fonction `dbopen()` renvoie un pointeur sur un objet de type **`DB`**, qui représente la base de données. Il s'agit d'une structure regroupant des méthodes d'accès à la manière des classes C++. Les membres qui nous intéressent sont tous des pointeurs sur des fonctions renvoyant une valeur `int`.

Nom	Arguments	Signification
`close`	`(const DB * db)`	Fermeture de la base de données.

`del`	`(const DB * db,`
	`const DBT * cle,`
	`int attributs)`
		Suppression de l'enregistrement correspondant à la clé indiquée ou de l'enregistrement à la position courante dans la base, si on transmet un attribut `R_CURSOR`.

`fd`	`(const DB * db)`	Obtention du descripteur de fichier associé à la base, sauf si celle-ci réside uniquement en mémoire.

`get`	`(const DB * db,`
	`const DBT * cle,`
	`DBT * donnee,`
	`int attributs)`
		Lecture de l'enregistrement correspondant à la clé transmise en argument. Les attributs ne sont pas utilisés, il faut mettre cet argument à 0.

`put`	`(const DB * db,`
	`const DBT * cle,`
	`const DBT * donnee,`
	`int attributs)`
		Pour enregistrer les données transmises. Si la clé existe déjà, l'enregistrement est écrasé, sauf si l'attribut `R_NOOVERWRITE` est employé. D'autres valeurs sont possibles pour les attributs, suivant le type de base de données.

`sync`	`(const DB * db,` `int attributs)`	Pour synchroniser les données en mémoire avec le fichier disque. L'attribut ne sert pas.

`seq`	`(const DB * db,`
	`DBT * cle,`
	`DBT * donnee,`
	`int attributs)`
		Recherche séquentielle dans la base de données. Avec l'attribut `R_FIRST`, on renvoie la première paire clé/donnée de la base, avec `R_NEXT` on renvoie la paire suivante.

Nous pouvons ainsi écrire un programme permettant de manipuler une base de données de manière générique. Ce logiciel acceptera les commandes suivantes :

- `put` : ajout d'un enregistrement ;
- `get` : recherche d'un enregistrement ;
- `del` : suppression d'un enregistrement ;
- `seq` : affichage du contenu de la base ;
- `quit` : fermer la base et quitter le programme.

exemple_dbopen.c :

```
#include <db_185.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
```

```
    #include <string.h>
    #include <sys/types.h>

    void traite_get (DB * db);
    void traite_put (DB * db);
    void traite_del (DB * db);
    void traite_seq (DB * db);

    int
main (int argc, char * argv[])
{
    DB *   db;
    DBTYPE dbtype;
    char   saisie[128];

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s fichier type \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (strcasecmp(argv[2], "btree") == 0)
        dbtype = DB_BTREE;
    else if (strcasecmp(argv[2], "hash") == 0)
        dbtype = DB_HASH;
    else if (strcasecmp(argv[2], "recno") == 0)
        dbtype = DB_RECNO;
    else {
        fprintf(stderr, "Types bases : btree, hash ou recno\n");
        exit(EXIT_FAILURE);
    }
    db = dbopen(argv[1], O_CREAT | O_RDWR, 0644, dbtype, NULL);
    if (db == NULL) {
        perror("dbopen");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "[commande]> ");
    while(fgets(saisie, 128, stdin) != NULL) {
        if (saisie[strlen(saisie) - 1] == '\n')
            saisie[strlen(saisie) - 1] = '\0';
        if (strcasecmp(saisie, "get") == 0)
            traite_get(db);
        else if (strcasecmp(saisie, "put") == 0)
            traite_put(db);
        else if (strcasecmp(saisie, "del") == 0)
            traite_del(db);
        else if (strcasecmp(saisie, "seq") == 0)
            traite_seq(db);
        else if (strncasecmp(saisie, "quit", 4) == 0)
            break;
        else
            fprintf(stdout, "Commandes : put, get, del, seq ou quit\n");
        fprintf(stdout, "[commande]> ");
    }
```

```
    db->close(db);
    return EXIT_SUCCESS;
}

    void
traite_get (DB * db)
{
    DBT    key;
    DBT    data;
    char   cle[128];
    char * donnee;
    int    retour;

    fprintf(stdout, "[clé]> ");
    if (fgets(cle, 128, stdin) == NULL) {
        fprintf(stdout, "Abandon !\n");
        return;
    }
    if (cle[strlen(cle) - 1] == '\n')
        cle[strlen(cle) - 1] = '\0';
    key.data = cle;
    key.size = strlen(cle) + 1;  /* avec '\0' */
    retour = db->get(db, & key, & data, 0);
    if (retour < 0)
        perror("get");
    if (retour > 0)
        fprintf(stdout, "Non trouvé\n");
    if (retour == 0) {
        donnee = malloc (data . size);
        if (donnee == NULL) {
            perror("malloc");
            return;
        }
        memcpy(donnee, data . data, data . size);
        fprintf(stdout, "%s\n", donnee);
        free(donnee);
    }
}

    void
traite_put (DB * db)
{
    DBT   key;
    DBT   data;
    char  cle [128];
    char  donnee [128];
    int   retour;

    fprintf(stdout, "[clé]> ");
    if (fgets(cle, 128, stdin) == NULL) {
        fprintf(stdout, "Abandon !\n");
```

```
        return;
    }
    if (cle[strlen(cle) - 1] == '\n')
        cle[strlen(cle) - 1] = '\0';
    key.data = cle;
    key.size = strlen(cle) + 1;
    fprintf(stdout, "[donnée]> ");
    if (fgets(donnee, 128, stdin) == NULL) {
        fprintf(stdout, "Abandon !\n");
        return;
    }
    if (donnee[strlen(donnee) - 1] == '\n')
        donnee[strlen(donnee) - 1] = '\0';
    data.data = donnee;
    data.size = strlen(donnee) + 1;
    retour = db->put(db, & key, & data, 0);
    if (retour < 0)
        perror("put");
    else
        fprintf(stdout, "Ok\n");
}

    void
traite_del (DB * db)
{
    DBT  key;
    char cle[128];
    int  retour;

    fprintf(stdout, "[clé]> ");
    if (fgets(cle, 128, stdin) == NULL) {
        fprintf(stdout, "Abandon !\n");
        return;
    }
    if (cle[strlen(cle) - 1] == '\n')
        cle[strlen(cle) - 1] = '\0';
    key.data = cle;
    key.size = strlen(cle) + 1;
    retour = db->del(db, & key, 0);
    if (retour < 0)
        perror("del");
    if (retour > 0)
        fprintf(stdout, "Non trouvé\n");
    if (retour == 0)
        fprintf(stdout, "Ok\n");
}

    void
traite_seq (DB * db)
{
    DBT  key;
    DBT  data;
```

```
    int  retour;

    for (retour = db->seq(db, & key, & data, R_FIRST);
         retour == 0;
         retour = db->seq(db, & key, & data, R_NEXT))
        fprintf(stdout, "%s\n    %s\n",
                (char *) key . data, (char *) data . data);
}
```

Ce programme va nous permettre de créer une petite base avec quelques chaînes de caractères et de les manipuler.

```
$ ./exemple_dbopen villes.btree btree
[commande]> put
[clé]> 1
[donnée]> BOURGES
Ok
[commande]> put
[clé]> 2
[donnée]> CHERBOURG
Ok
[commande]> put
[clé]> 3
[donnée]> DIEPPE
Ok
[commande]> put
[clé]> 4
[donnée]> EPERNAY
Ok
[commande]> seq
1
    BOURGES
2
    CHERBOURG
3
    DIEPPE
4
    EPERNAY
 [commande]> get
[clé]> 7
Non trouvé
[commande]> get
[clé]> 3
DIEPPE
[commande]> del
[clé]> 2
Ok
[commande]> get
[clé]> 2
Non trouvé
[commande]> seq
```

```
1
    BOURGES
3
    DIEPPE
4
    EPERNAY
[commande]> quit
$ ls -l villes.btree
-rw-r--r--   1 ccb    ccb      8192 Feb 16 15:34 villes.btree
$
```

Conclusion

Nous n'avons présenté ici que le minimum vital pour manipuler les bases de données DB Berkeley. Il existe des fonctions bien plus complètes, supportant la notion de transaction et le positionnement de curseurs par exemple. On trouvera des renseignements dans la documentation disponible sur le site web de *Sleepycat Software*.

Avec l'étude des bases de données, nous achevons une partie consacrée à l'ensemble des routines permettant de gérer des fichiers, avec des formes très diverses. Nous allons maintenant nous intéresser pendant quelques chapitres aux données proprement dites, en examinant les conversions de type, les routines mathématiques et les informations disponibles sur le système.

23

Types de données et conversions

Types de données génériques

Les types de données connus par le compilateur C sous Linux sont les suivants :

`char`, `short int`, `int`, `long int`, `long long int`, `float`, `double`, `long double` et `void*`. On peut y ajouter les variantes `unsigned` des types entiers, mais elles ont la même taille que leur équivalent `signed`. Le type `long long int` est une extension par rapport au C Ansi.

La taille nécessaire pour stocker les données est déterminée à l'aide de la fonction **`sizeof()`**. On notera qu'il ne s'agit pas d'une fonction de bibliothèque mais d'un opérateur du langage C appartenant à la liste de ses mots-clés, au même titre que `for`, `if`, `switch`…

Voici la taille des données génériques sur un PC sous Linux, avec les options standard du compilateur :

Type	Taille (en octets)
`char`	1
`short int`	2
`int`	4
`long int`	4
`long long int`	8
`float`	4
`double`	8
`long double`	12
`void *`	4

Pour certains types entiers, les valeurs minimale et maximale sont définies sous forme de constantes symboliques dans `<limits.h>`. Il y a cette fois une différence entre les types entiers signés et non signés. Bien entendu, les types non signés commencent tous à 0. Voici les noms des constantes symboliques représentant les limites, ainsi que leurs valeurs sur un PC avec un processeur 32 bits :

Type	Nom limite	Valeur limite
`signed char`	`SCHAR_MIN`	–128
	`SCHAR_MAX`	127
`unsigned char`	`UCHAR_MAX`	255
`signed short int`	`SHRT_MIN`	–32 768
	`SHRT_MAX`	32 767
`unsigned short int`	`USHRT_MAX`	65 535
`signed int`	`INT_MIN`	–2 147 483 648
	`INT_MAX`	2 147 483 647
`unsigned int`	`UINT_MAX`	4 294 967 295
`signed long int`	`LONG_MIN`	–2 147 483 648
	`LONG_MAX`	2 147 483 647
`unsigned long int`	`ULONG_MAX`	4 249 967 295

À partir de ces types, la bibliothèque C définit, par `typedef` ou `#define`, tous les types spécifiques qu'on peut rencontrer, comme `size_t`, `time_t`, etc. Certains d'entre eux sont des structures, comme `rusage` que nous avons vue dans le chapitre 5, ou des unions, comme `sigval` que nous avons rencontrée dans le chapitre 8.

Catégories de caractères

Les caractères représentent le bloc fondamental sur lequel repose tout le dialogue avec l'utilisateur. Un programme peut manipuler en interne des entiers, des réels, ou même des objets structurés complexes, mais dans tous les cas les saisies et les affichages se feront par l'intermédiaire de caractères. Il est donc normal qu'il existe une quinzaine de fonctions faisant partie du C Ansi, permettant de préciser l'appartenance d'un caractère à une ou plusieurs catégories bien définies. Ces fonctions permettent par exemple de s'assurer qu'un caractère est bien une majuscule, un chiffre, un symbole affichable, etc.

Le prototype général de ces routines, déclarées dans `<ctype.h>`, est le suivant :

```
int is<TYPE> (int caractere);
```

La valeur passée en argument doit correspondre à celle d'une donnée de type `char` ou à la rigueur à la valeur `EOF`. Ceci permet de traiter directement la sortie d'une routine comme `getchar()`.

Une routine de cet ensemble est particulièrement précieuse pour analyser le résultat d'une fonction de saisie ou pour afficher correctement des données binaires, comme nous l'avons fait dans le programme `exemple_getchar.c` du chapitre 10.

Nom	Type de caractères
`isalnum()`	Caractère alphanumérique, lettre ou chiffre.
`isalpha()`	Caractère alphabétique. Dans la localisation C par défaut, il s'agit uniquement des lettres `A-Z` et `a-z` sans accentuation.
`isascii()`	Caractère appartenant au standard Ascii (compris entre 0 et 127). La table Ascii est rappelée en annexe.
`isblank()`	Caractère blanc, c'est-à-dire un espace ou une tabulation.
`iscntrl()`	Caractère de contrôle non imprimable.
`isdigit()`	Chiffre décimal.
`isgraph()`	Caractère imprimable ayant un symbole non blanc.
`islower()`	Lettre minuscule. Dans la localisation C par défaut, les minuscules accentuées ne sont pas comprises dans cette catégorie.
`isprint()`	Caractère imprimable, c'est-à-dire un caractère graphique ou un espace.
`ispunct()`	Caractère de ponctuation. Ceci recouvre les caractères graphiques non alphanumériques.
`isspace()`	Caractère d'espacement comprenant par exemple les tabulations horizontale et verticale, le saut de ligne, le retour chariot ou le saut de page.
`isupper()`	Caractère majuscule.
`isxdigit()`	Chiffre hexadécimal.

Attention

Les routines `isTYPE()` comme les trois routines `toTYPE()`, que nous verrons dans la prochaine section, peuvent être implémentées – dans d'anciennes bibliothèques C – sous forme de macros définies dans `<ctype.h>`, évaluant plusieurs fois leurs arguments.

Il faut donc éviter tout effet de bord, comme dans

```
while (i < strlen(saisie))
    if (! isdigit(saisie[i++]))
        return -1;
```

qui risque de ne vérifier qu'un caractère sur deux si `isdigit()` est implémentée ainsi :

```
#define isdigit(x)  ((x >= '0') && (x <= '9'))
```

Le programme suivant permet d'examiner les caractéristiques des caractères saisis en entrée.

exemple_is.c :

```
    #include <ctype.h>
    #include <locale.h>
    #include <stdio.h>

    void
affiche_caracteristiques (int c)
{
    fprintf(stdout, "%02X : ", (unsigned char) c);
    if (isalnum(c))    fprintf(stdout, "alphanumérique ");
```

```
    if (isalpha(c))    fprintf(stdout, "alphabétique ");
    if (isascii(c))    fprintf(stdout, "ascii ");
    if (iscntrl(c))    fprintf(stdout, "contrôle ");
    if (isdigit(c))    fprintf(stdout, "chiffre ");
    if (isgraph(c))    fprintf(stdout, "graphique ");
    if (islower(c))    fprintf(stdout, "minuscule ");
    if (isprint(c))    fprintf(stdout, "imprimable ");
    if (ispunct(c))    fprintf(stdout, "ponctuation ");
    if (isspace(c))    fprintf(stdout, "espace ");
    if (isupper(c))    fprintf(stdout, "majuscule ");
    if (isxdigit(c))   fprintf(stdout, "hexadécimal ");
    fprintf(stdout, "\n");
}

    int
main (void)
{
    char chaine[128];
    int  i;
    setlocale(LC_ALL, "");
    while (fgets(chaine, 128, stdin) != NULL)
        for (i = 0; i < strlen(chaine); i ++)
            affiche_caracteristiques(chaine[i]);
    return EXIT_SUCCESS;
}
```

Nous allons observer les effets de la localisation sur ces fonctions, en commençant par utiliser la localisation par défaut.

```
$ unset LC_ALL
$ unset LANG
$ ./exemple_is
az 1    é
61 : alphanumérique alphabétique ascii graphique minuscule imprimable hexadécimal
7A : alphanumérique alphabétique ascii graphique minuscule imprimable
20 : ascii imprimable espace
31 : alphanumérique ascii chiffre graphique imprimable hexadécimal
09 : ascii contrôle espace
E9 :
0A : ascii contrôle espace
  (Contrôle - D)
$
```

Nous remarquons que le caractère `a` (61) est considéré comme une lettre mais aussi comme un chiffre hexadécimal, ce qui n'est pas le cas de `z` (7A). L'espace (20) est imprimable, alors que la tabulation (09) entre le `1` et le `é` est considérée comme un caractère de contrôle, au même titre que le retour chariot (0A) en fin de saisie.

Le cas du caractère `é` (E9) est plus surprenant. Comme on s'y attendait, il n'est pas considéré comme un caractère Ascii car son code est supérieur à 127. Il n'est pas vu non plus comme une lettre puisque dans la localisation par défaut elles sont toutes dans la table Ascii. Ce qui est encore plus étonnant, c'est qu'il n'est même pas considéré comme un caractère imprimable.

En fait c'est logique, car la partie supérieure de la table des caractères n'est pas définie si aucune localisation n'est choisie. Le code E9 n'est donc associé à aucun symbole particulier. La correspondance E9 – é est assurée ici uniquement par le terminal. Par contre, si nous définissons la localisation correctement, le comportement est différent :

```
$ export LANG=fr_FR
$ ./exemple_is
éàË
E9 : alphanumérique alphabétique graphique minuscule imprimable
E0 : alphanumérique alphabétique graphique minuscule imprimable
CB : alphanumérique alphabétique graphique imprimable majuscule
0A : ascii contrôle espace
  (Contrôle - D)
$
```

Les caractères accentués sont à présent reconnus non seulement comme des lettres, mais leur classification en majuscules et minuscules est également correcte.

Conversion entre catégories de caractères

Les conversions de caractères entre différentes catégories sont très limitées. Il existe trois fonctions permettant de modifier la classe d'un caractère, **toascii()**, **toupper()** et **tolower()**, dont les prototypes sont déclarés dans `<ctype.h>` :

```
int toascii (int caractere);
int toupper (int caractere);
int tolower (int caractere);
```

La fonction `toascii()` supprime purement et simplement le huitième bit du caractère transmis afin de renvoyer une valeur comprise entre 0 et 127. On comprend bien que le caractère résultant de cette modification n'a que très peu de chance d'avoir quelque chose à voir avec la lettre originale. En particulier, un caractère accentué comme `é` n'est pas transformé en `e` mais en un caractère quelconque de la table Ascii – en l'occurrence `i`. Ceci explique les modifications parfois étranges des textes contenant des caractères accentués lorsqu'ils franchissent des passerelles de courrier électronique mal configurées.

Les fonctions `toupper()` et `tolower()` permettent respectivement de passer un caractère en majuscule et en minuscule. Ces fonctions sont sensibles à la localisation. Ainsi `toupper('é')` renverra le caractère `É` dans une localisation `fr_FR` par exemple.

> **Attention**
> Dans les bibliothèques C courantes, `toupper()` ne modifie pas le caractère passé en argument si ce n'est pas une minuscule. Mais dans des versions plus anciennes, cette routine renvoyait un caractère erroné car elle modifiait toujours le sixième bit de la lettre. Ceci est également vrai avec `tolower()` et les caractères non majuscules.

On emploie donc systématiquement une vérification du genre :

```
if (isupper(c))
    c = tolower(c);
```

ou

```
if (islower(c))
    c = toupper(c);
```

Conversions de données entre différents types

Les conversions qui nous intéressent ici sont celles qui permettent de passer d'une valeur numérique entière ou réelle à une chaîne de caractères, et inversement. Les conversions mathématiques entre réels et entiers seront abordées dans le prochain chapitre.

Il est toujours possible d'utiliser `sprintf()` ou `sscanf()` pour présenter les résultats d'un calcul ou examiner le contenu d'une chaîne, comme nous l'avons vu dans le chapitre 10. Toutefois, le surcoût imposé par l'énorme machine que représente `sscanf()` ne se justifie pas lorsqu'on veut juste convertir une chaîne de trois caractères en une valeur numérique comprise entre 1 et 100. Si la conversion n'a lieu qu'une seule fois avant un gros calcul et une fois après pour afficher le résultat, mieux vaut probablement employer `sscanf()` et `sprintf()`, dont on maîtrise généralement mieux l'interface. Néanmoins, si on doit convertir à répétition les coordonnées de 200 000 points contenues dans des chaînes de caractères alors que l'utilisateur attend le résultat, il est sûrement préférable d'employer des routines optimisées.

Pour ce genre d'opération, il existe des fonctions spécialisées très efficaces. Les plus simples sont **`atoi()`**, **`atol()`**, **`atof()`**, ainsi que **`atoll()`**. Déclarées dans `<stdlib.h>`, ces routines convertissent les chaînes de caractères passées en arguments dans les types correspondant à leurs noms :

```
int atoi (const char * chaine);
long atol (const char * chaine);
long long atoll (const char * chaine);
double atof (const char * chaine);
```

Le problème que posent ces routines réside dans l'impossibilité de déterminer si une erreur s'est produite, comme le montre le programme suivant :

exemple_atoi.c :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char chaine[128];
    while (fgets(chaine, 128, stdin) != NULL)
        fprintf(stdout, "Lu : %d \n", atoi(chaine));
    return EXIT_SUCCESS;
}
```

La routine ne permet pas de faire la différence entre 0 et une chaîne invalide :

```
$ ./exemple_atoi
4767
Lu : 4767
-101325
```

```
Lu : -101325
-2147483648
Lu : -2147483648
-2147483649
Lu : -2147483648
-2200000000
Lu : -2147483648
0
Lu : 0
azerty
Lu : 0
  (Contrôle-D)
$
```

On remarque que la fonction plafonne les valeurs à la limite du type de données correspondant. Toutefois, le fait de ne pas pouvoir détecter des situations d'erreur est très dangereux, aussi évitera-t-on au maximum d'employer ces routines, à moins d'avoir auparavant vérifié entièrement le contenu de la chaîne.

Il est souvent préférable de se tourner vers les fonctions **strtol()**, **strtoul()**, **strtoll()** et **strtoull()**, déclarées dans `<stdlib.h>` :

```
long int           strtol   (const char * chaine, char ** fin, int base);
unsigned long int  strtoul  (const char * chaine, char ** fin, int base);
long long int      strtoll  (const char * chaine, char ** fin, int base);
unsigned long long strtoull (const char * chaine, char ** fin, int base);
```

Ces fonctions analysent la chaîne de caractères passée en premier argument et en extraient une variable entière qu'elles retournent. Le second argument, s'il n'est pas nul, est un pointeur qui est mis à jour pour être dirigé vers le premier caractère non utilisé par la conversion. Finalement, le dernier argument représente la base employée pour la lecture. La base peut s'étendre de 2 à 36 ou prendre la valeur spéciale 0. Alors, la lecture est effectuée en base 10, sauf si la chaîne commence par `0x`, cas où la conversion sera en hexadécimal, ou par un `0`, cas où la lecture se fera en octal.

Pour les bases supérieures à 10, on emploie les lettres dans l'ordre alphabétique pour compléter les chiffres manquants. Ainsi, on utilise `A`, `B`, `C`, `D`, `E` et `F` en hexadécimal, et toutes les lettres jusqu'à `Z` en base 36. Il n'y a pas de différences entre les majuscules et les minuscules. Tous les caractères d'espacement en début de chaîne sont ignorés.

Le pointeur fourni en second argument permet de savoir si la conversion a pu avoir lieu. En effet, si aucun chiffre n'est lu, `*fin` est égal à `chaine`. En cas de débordement supérieur ou inférieur, la valeur renvoyée est plafonnée à la limite maximale ou supérieure du type de donnée, et `errno` vaut `ERANGE`. Voici un exemple d'utilisation de `strtol()`.

exemple_strtol.c :

```
    #include <errno.h>
    #include <limits.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
```

```
{
    char   chaine[128];
    char * fin;
    long   retour;
    fprintf(stdout, "> ");
    while (fgets(chaine, 128, stdin) != NULL) {
        retour = strtol(chaine, & fin, 0);
        if (fin == chaine) {
            fprintf(stdout, "  Erreur \n> ");
            continue;
        }
        if (((retour == LONG_MAX) || (retour == LONG_MIN))
         && (errno == ERANGE)) {
            fprintf(stdout, "  Débordement ! \n> ");
            continue;
        }
        fprintf(stdout, "  Lu : %ld \n> ", retour);
    }
    return EXIT_SUCCESS;
}
```

Nous pouvons observer que, cette fois, la détection d'erreur est parfaitement gérée :

```
$ ./exemple_strtol
> 0xFFFF
  Lu : 65535
> -2147483648
  Lu : -2147483648
> -2147483649
  Débordement !
> 99999999999
  Débordement !
> azerty
  Erreur
> 0
  Lu : 0
>  (Contrôle-D)
$
```

Pour lire des valeurs réelles, il existe trois fonctions **strtod()**, **strtof()** et **strtold()**.

```
float       strtof  (const char * chaine, char ** fin);
double      strtod  (const char * chaine, char ** fin);
long double strtold (const char * chaine, char ** fin);
```

Ces routines fonctionnent comme leurs consœurs entières, à la différence qu'il n'y a pas de notion de base ici, toutes les représentations étant considérées comme décimales. De plus, les routines de conversion de réels sont sensibles à la localisation pour tout ce qui concerne le séparateur décimal. Dans l'exemple précédent nous n'avons pas utilisé la possibilité de lire successivement plusieurs valeurs au sein de la même chaîne. Dans le programme suivant nous allons nous y employer.

exemple_strtof.c :

```
    #define _GNU_SOURCE

    #include <errno.h>
    #include <limits.h>
    #include <locale.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    char   chaine[128];
    char * debut;
    char * fin;
    float  retour;

    setlocale(LC_ALL, "");
    while (fgets(chaine, 128, stdin) != NULL) {
        if (chaine[strlen(chaine) - 1] == '\n')
            chaine[strlen(chaine) - 1] = '\0';
        for (fin = debut = chaine; * fin != '\0'; debut = fin) {
            errno = 0;
            retour = strtof(debut, & fin);
            if (fin == debut) {
                fprintf(stdout, "Erreur \n");
                break;
            }
            if (errno == ERANGE)
                fprintf(stdout, "Débordement ! \n");
            else
                fprintf(stdout, "Lu : %f \n", retour);
        }
    }
    return EXIT_SUCCESS;
}
```

Commençons par vérifier le comportement vis-à-vis de la localisation :

```
$ unset LC_ALL
$ unset LANG
$ ./exemple_strtof
1.5
Lu : 1.500000
1,8
Lu : 1.000000
Erreur
$ export LC_ALL=fr_FR
$ ./exemple_strtof
1.5
Lu : 1,000000
Erreur
```

```
1,8
Lu : 1,800000
  (Contrôle-D)
$
```

Nous observons au passage que `printf()` est également sensible à la localisation pour ce qui concerne l'affichage de la valeur réelle. À présent, vérifions le problème du débordement :

```
$ ./exemple_strtof
9999999999
Lu : 10000000000,000000
1e10
Lu : 10000000000,000000
1e20
Lu : 100000002004087734272,000000
1e30
Lu : 1000000015047466219876688855040,000000
1e40
Débordement !
1e39
Débordement !
1e38
Lu : 999999968028569246506562607691732090 88,000000
  (Contrôle-D)
$
```

Nous constatons par la même occasion que la précision d'une variable `float` est assez limitée. Nous pouvons aussi examiner le fonctionnement des lectures successives dans la même chaîne :

```
$ ./exemple_strtof
04 07 67
Lu : 4,000000
Lu : 7,000000
Lu : 67,000000
30 07 68 azerty
Lu : 30,000000
Lu : 7,000000
Lu : 68,000000
Erreur
$
```

Parallèlement à ces fonctions de lecture de chaîne, il existe des routines spécialisées dans la conversion de variables réelles en chaînes de caractères. Étant peu portables et compliquées à utiliser, on les déconseille en général. Il est souvent préférable d'employer `sprintf()`.

Les routines **`ecvt()`**, **`fcvt()`** et **`gcvt()`** sont héritées de Système V. Leurs prototypes sont déclarés dans `<stdlib.h>` ainsi :

```
char * ecvt (double nombre, size_t nb_chiffres,
             int * position_point, int * signe);
char * fcvt (double nombre, size_t nb_chiffres,
             int * position_point, int * signe);
char * gcvt (double nombre, size_t nb_chiffres,
             char * buffer);
```

La fonction `ecvt()` convertit la valeur passée en premier argument en une chaîne de caractères contenant au maximum le nombre de chiffres indiqué en second argument. La chaîne renvoyée est allouée dans la mémoire statique – écrasée à chaque appel – et ne comprend pas de point décimal. En contrepartie, le troisième argument comportera en retour la position du premier chiffre après ce point décimal. Enfin, le dernier argument sera rempli avec une valeur nulle si le chiffre est positif.

La routine `fcvt()` fonctionne de la même manière, mais le second argument indique le nombre de décimales désirées.

La fonction `gcvt()` écrit le nombre de chiffres significatifs indiqué en second argument dans le buffer qui est passé en troisième argument, et renvoie un pointeur sur celui-ci.

L'utilisation de `ecvt()` et `fcvt()` est loin d'être intuitive. En voici un exemple :

```
    #include <stdio.h>
    #include <stdlib.h>

    int
main (int argc, char * argv[])
{
    double valeur;
    int    nb_chiffres;
    int    position;
    int    signe;
    char * retour;

    if ((argc != 3)
     || (sscanf(argv[1], "%lf", & valeur) != 1)
     || (sscanf(argv[2], "%d", & nb_chiffres) != 1)) {
        fprintf(stderr, "Syntaxe : %s valeur nb_chiffres \n",
        argv[0]);
        exit(EXIT_FAILURE);
    }
    retour = ecvt(valeur, nb_chiffres, & position, & signe);
    fprintf(stdout, "ecvt() = %s \n", retour);
    fprintf(stdout, "    position = %d \n", position);
    fprintf(stdout, "    signe = %d \n", signe);

    retour = fcvt(valeur, nb_chiffres, & position, & signe);
    fprintf(stdout, "fcvt() = %s \n", retour);
    fprintf(stdout, "    position = %d \n", position);
    fprintf(stdout, "    signe = %d \n", signe);

    return EXIT_SUCCESS;
}
```

Les exécutions suivantes montrent bien que prévoir le résultat de ces routines nécessite une bonne dose de concentration :

```
$ ./exemple_ecvt 100 3
ecvt() = 100
    position = 3
    signe = 0
```

```
fcvt() = 100000
    position = 3
    signe = 0
$ ./exemple_ecvt 100 2
ecvt() = 10
    position = 3
    signe = 0
fcvt() = 10000
    position = 3
    signe = 0
$ ./exemple_ecvt 1.5 3
ecvt() = 150
    position = 1
    signe = 0
fcvt() = 1500
    position = 1
    signe = 0
$ ./exemple_ecvt -1.5 2
ecvt() = 15
    position = 1
    signe = 1
fcvt() = 150
    position = 1
    signe = 1
$
```

La bibliothèque Gnu ajoute en extensions les fonctions **`qecvt()`**, **`qfcvt()`** et **`qgcvt()`**, qui ont un comportement similaire mais en utilisant des valeurs `long double` (*quad*).

```
char * qecvt (long double nombre, size_t nb_chiffres,
              int * position_point, int * signe);
char * qfcvt (long double nombre, size_t nb_chiffres,
              int * position_point, int * signe);
char * qgcvt (long double nombre, size_t nb_chiffres, char * buffer);
```

Enfin, toutes ces fonctions renvoyant leurs valeurs dans des zones de mémoire statique, elles ne sont pas utilisables dans un contexte multithread. Il existe donc quatre autres extensions Gnu, **`ecvt_r()`**, **`fcvt_r()`**, **`qecvt_r()`** et **`qfcvt_r()`**, auxquelles on transmet un buffer personnel à remplir en indiquant sa taille maximale.

```
char * ecvt_r  (double nombre, size_t nb_chiffres,
                int * position_point, int * signe,
                char * buffer, size_t longueur);
char * fcvt_r  (double nombre, size_t nb_chiffres,
                int * position_point, int * signe,
                char * buffer, size_t longueur);
char * qecvt_r (long double nombre, size_t nb_chiffres,
                int * position_point, int * signe,
                char * buffer, size_t longueur);
char * qfcvt_r (long double nombre, size_t nb_chiffres,
                int * position_point, int * signe,
                char * buffer, size_t longueur);
```

Les routines `gcvt()` et `qgcvt()` emploient déjà un buffer transmis par l'application et ne nécessitent donc pas d'équivalentes réentrantes. Rappelons que ces fonctions sont difficiles à employer et qu'il vaut généralement mieux se tourner vers `sprintf()` qui peut offrir les mêmes résultats. Pour cette raison, d'ailleurs, `ecvt()`, `fcvt()` et `gcvt()` ont été supprimées du standard ISO C9X.

Caractères étendus

L'internationalisation des programmes est devenue, principalement depuis le développement exponentiel des accès Internet, une priorité pour de nombreux développeurs. Les applications sont longtemps restées cantonnées dans l'emploi de jeux de caractères limités, tels que l'Ascii ou ses extensions ISO (comme les ensembles ISO 8859-1 et ISO 8859-15 présentés en annexe). Toutefois, il existe de nombreuses langues dont l'alphabet ne peut pas tenir sur une table de 255 caractères. Pour résoudre ce problème, on a introduit le principe des caractères larges de type **`wchar_t`** (*wide characters*). Ceux-ci suivent les normes de représentation *ISO-10646* et son sous-ensemble *Unicode*, qui regroupent quasiment l'ensemble des alphabets connus.

Une application manipulant des chaînes composées de caractères larges offre une garantie de portabilité au niveau des textes traités. Le type `wchar_t` peut être comparé au `char` original, étendu sur un nombre plus important de bits (31 en général).

Pour offrir une symétrie parfaite avec les fonctions manipulant des caractères normaux, il existe un type **`wint_t`**, capable de recevoir n'importe quel caractère large, ainsi que la constante particulière **`WEOF`**.

Deux constantes symboliques, définies dans `<wchar.h>`, permettent de connaître les limites des objets de type `wchar_t` : **`WCHAR_MIN`** et **`WCHAR_MAX`**.

Pour indiquer au compilateur qu'une constante doit être considérée comme un caractère large, on utilise le préfixe `L`. Ainsi on écrira :

```
wchar_t chaine [127];
chaine [0] = L'\0' ;
```

ou

```
if (reponse_saisie [0] == L'N') || (reponse_saisie [0] == L'n')
    return (-1);
```

Naturellement, de nouvelles fonctions doivent être introduites pour offrir les mêmes possibilités de manipulation des caractères et des chaînes larges que celles dont nous disposions déjà avec les caractères simples. Les fonctions de manipulation des chaînes de caractères larges sont déclarées dans `<wchar.h>`, en remplaçant simplement les chaînes `char *` en `wchar_t *`. Un caractère nul large `L'\0'` sert à indiquer la fin de la chaîne.

Fonction avec chaînes larges	Fonction équivalente avec chaînes simples
`wcslen (wchar_t * chaine);`	`strlen (char * chaine);`
`wcsnlen (wchar_t * chaine, size_t * maximum);`	`strnlen (char * chaine, size_t maximum);`
`wcscpy (wchar_t * cible, wchar_t * source);`	`strcpy (char * cible, char * source);`

Fonction avec chaînes larges	Fonction équivalente avec chaînes simples
`wcsncpy (wchar_t * cible, wchar_t * source, size_t taille);`	`strncpy (char * cible, char * source, size_t taille);`
`wcscat (wchar_t * cible, wchar_t * source);`	`strcat (char * cible, char * source);`
`wcsncat (wchar_t * cible, wchar_t * source, size_t taille);`	`strncat (char * cible, char * source, size_t taille);`
`wcscmp (wchar_t * chaine_1, wchar_t * chaine_2);`	`strcmp (char * chaine_1, char * chaine_2);`
`wcsncmp (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);`	`strncmp (char * chaine_1, char * chaine_2, size_t taille);`
`wcscasecmp (wchar_t * chaine_1, wchar_t * chaine_2);`	`strcasecmp (char * chaine_1, char * chaine_2);`
`wcsncasecmp (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);`	`strncasecmp (char * chaine_1, char * chaine_2, size_t taille);`
`wcscoll (wchar_t * chaine_1, wchar_t * chaine_2);`	`strcoll (char * chaine_1, char * chaine_2);`
`wcsxfrm (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);`	`strxfrm (char * chaine_1, char * chaine_2, size_t taille);`
`wcschr (wchar_t * chaine, wchar_t caractere);`	`strchr (char * chaine, char caractere);`
`wcsrchr (wchar_t * chaine, wchar_t caractere);`	`strrchr (char * chaine, char caractere);`
`wcscspn (wchar_t * chaine, wchar_t * ensemble);`	`strcspn (char * chaine, char * ensemble);`
`wcsspn (wchar_t * chaine, wchar_t * ensemble);`	`strspn (char * chaine, char * ensemble);`
`wcspbrk (wchar_t * chaine, wchar_t * ensemble);`	`strpbrk (char * chaine, char * ensemble);`
`wcsstr (wchar_t * chaine, wchar_t * sous_chaine);`	`strstr (char * chaine, char * sous_chaine);`
`wcstok (wchar_t * chaine, wchar_t * separateurs, wchar_t ** pointeur);`	`strtok (char * chaine, char * separateurs, char ** pointeur);`
`wmemchr (wchar_t * chaine, wchar_t caractere, size_t taille);`	`memchr (char * chaine, char caractere, size_t taille);`

Fonction avec chaînes larges	Fonction équivalente avec chaînes simples
`wmemset (wchar_t * chaine, wchar_t caractere, size_t taille);`	`memset (char * chaine, char caractere, size_t taille);`
`wmemcmp (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);`	`memcmp (char * chaine_1, char * chaine_2, size_t taille);`
`wmemcpy (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);`	`memcpy (char * chaine_1, char * chaine_2, size_t taille);`
`wmemmove (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);`	`memmove (char * chaine_1, char * chaine_2, size_t taille);`
`wcstod (wchar_t * chaine, wchar_t ** fin)`	`strtod (char * chaine, char ** fin);`
`wcstof (wchar_t * chaine, wchar_t ** fin);`	`strtof (char * chaine, char ** fin);`
`wcstol (wchar_t * chaine, wchar_t ** fin, int base);`	`strtol (char * chaine, char ** fin, int base);`
`wcstold (wchar_t * chaine, wchar_t ** fin);`	`strtold (char * chaine, char ** fin);`
`wcstoll (wchar_t * chaine, wchar_t ** fin, int base);`	`strtoll (char * chaine, char ** fin, int base);`
`wcstoul (wchar_t * chaine, wchar_t ** fin, int base);`	`strtoul (char * chaine, char ** fin, int base);`
`wcstoull (wchar_t * chaine, wchar_t ** fin, int base);`	`strtoull (char * chaine, char ** fin, int base);`

Nous voyons que les noms des fonctions remplacent le préfixe `str` (*string*) par `wcs` (*wide char string*), mais que les possibilités restent les mêmes.

Pour manipuler des caractères larges seuls, on peut utiliser des routines équivalentes à celles que nous avons rencontrées au début de ce chapitre, déclarées dans `<wctype.h>` :

Fonction avec caractère large	Fonction équivalente avec caractère simple
`iswalnum (wint_t caractere);`	`isalnum (int caractere);`
`iswalpha (wint_t caractere);`	`isalpha (int caractere);`
`iswblank (wint_t caractere);`	`isblank (int caractere);`
`iswcntrl (wint_t caractere);`	`iscntrl (int caractere);`

Fonction avec caractère large	Fonction équivalente avec caractère simple
`iswdigit (wint_t caractere);`	`isdigit (int caractere);`
`iswgraph (wint_t caractere);`	`isgraph (int caractere);`
`iswlower (wint_t caractere);`	`islower (int caractere);`
`iswprint (wint_t caractere);`	`isprint (int caractere);`
`iswpunct (wint_t caractere);`	`ispunct (int caractere);`
`iswspace (wint_t caractere);`	`isspace (int caractere);`
`iswupper (wint_t caractere);`	`isupper (int caractere);`
`iswxdigit (wint_t caractere);`	`isxdigit (int caractere);`
`towlower (wint_t caractere);`	`tolower (int caractere);`
`towupper (wint_t caractere);`	`toupper (int caractere);`

Enfin, pour pouvoir assurer des entrées-sorties employant des chaînes de caractères larges, on ajoute quelques spécifications de type aux formats employés par `printf()` et `scanf()` :

Conversion	Signification
`%C`	Caractère large : `printf()` attend un argument de type `wchar_t`, alors que `scanf()` nécessite un pointeur sur un caractère large.
`%S`	Chaîne de caractères larges : `printf()` comme `scanf()` demandent un argument de type `wchar_t *`.

Nous voyons qu'une application peut donc aisément manipuler des caractères larges, permettant de disposer d'une ouverture vers l'ensemble des alphabets du globe.

Les routines spécifiques sont définies dans `<wchar.h>` :

Nouvelle routine caractères larges	Routine équivalente caractères simples
`wint_t fgetwc (FILE * flux);`	`int fgetc (FILE * flux);`
`wint_t getwc (FILE * flux);`	`int getc (FILE * flux);`
`wint_t getwchar(void)`	`int getchar(void)`
`wchar_t * fgetws(wchar_t * chaine, size_t taille, FILE * flux);`	`char * fgets (char * chaine, size_t taille, FILE * flux);`
`wint_t fputwc (wchar_t caractere, FILE * flux);`	`int fputc (char caractere, FILE * flux);`
`wint_t putwc (wchar_t caractere, FILE * flux);`	`int putc (char caractere, FILE * flux);`
`wint_t putwchar(wchar_t caractere)`	`int putchar(char caractere)`
`wint_t fputws (wchar_t * chaine, FILE * flux);`	`int fputs (char * chaine, FILE * flux);`

Nouvelle routine caractères larges	Routine équivalente caractères simples
`wint_t ungetwc (wint_t caractere, FILE * flux);`	`int ungetc (int caractere, FILE * flux);`
`int wprintf (wchar_t * format, ...);`	`int printf (char * format, ...);`
`int fwprintf (FILE * flux, wchar_t * format, ...);`	`int fprintf (FILE * flux, char * format, ...);`
`int swprintf (wchar_t * cible, size_t maximum, wchar_t * format, ...);`	`int sprintf (char * cible, size_t maximum, char * format, ...);`
`int vwprintf (wchar_t * format, va_list args);`	`int vprintf (char * format, va_list args);`
`int vfwprintf (FILE * flux, wchar_t * format, va_list args);`	`int vfprintf (FILE * flux, char * format, va_list args);`
`int vswprintf (wchar_t * cible, size_t maximum, wchar_t * format, va_list args);`	`int vsprintf (wchar_t * cible, size_t maximum, char * format, va_list args);`
`int wscanf (wchar_t * format, ...);`	`int scanf (char * format, ...);`
`int fwscanf (FILE * flux, wchar_t * format, ...);`	`int fscanf (FILE * flux, char * format, ...);`
`int swscanf (wchar_t * cible, size_t maximum, wchar_t * format, ...);`	`int sscanf (char * cible, size_t maximum, char * format, ...);`
`int vwscanf (wchar_t * format, va_list args);`	`int vscanf (char * format, va_list args);`
`int vfwscanf (FILE * flux, wchar_t * format, va_list args);`	`int vfscanf (FILE * flux, char * format, va_list args);`
`int vswscanf (wchar_t * contenu, size_t maximum, wchar_t * format, va_list args);`	`int vsscanf (char * contenu, size_t maximum, char * format, va_list args);`

Caractères étendus et séquences multioctets

Une application peut donc manipuler, nous l'avons vu, des chaînes de caractères larges en interne, et l'interface avec l'utilisateur est également définie, même si elle n'est pas totalement implémentée sous Linux. Toutefois, un problème se pose pour l'échange de données entre applications différentes.

Non seulement la représentation interne des caractères larges est théoriquement opaque, mais même lorsqu'on la connaît, elle est dépendante par exemple de l'ordre des octets sur la machine. Pour transférer des données entre applications et entre systèmes différents, on emploie une autre représentation : les séquences multioctets.

Dans ce cadre, les caractères sont manipulés comme des chaînes d'octets dont la taille peut varier suivant le caractère considéré. Le standard UTF-8 qui est employé sous Linux pour l'encodage des caractères en séquence multioctet est très économe. Les caractères Ascii classiques (inférieurs à 128) sont représentés par un seul octet. Les caractères UCS inférieurs à 2 048 tiennent sur deux octets, et ainsi de suite jusqu'à un maximum de 6 octets pour couvrir tout l'espace UCS de 31 bits. Les caractères les plus employés dans les communications internationales conservent donc un encombrement minimal.

De plus, les caractères n'appartenant pas à la table Ascii sont représentés par des séquences d'octets compris entre 128 et 253. Un caractère inférieur à 128 ne peut donc être qu'un caractère Ascii. Il n'y a donc pas d'ambiguïté, on ne risque pas d'introduire involontairement des caractères de contrôle, des séparateurs de chemin comme '/', ni surtout un caractère nul dans le corps d'une chaîne. Le standard UTF-8 est donc directement utilisable au niveau du système pour représenter des chemins d'accès, des noms de machines, etc.

Cette représentation est largement employée, mais elle n'est pas unique. La bibliothèque C peut décider d'utiliser des conversions différentes, en fonction de la localisation par exemple, et il faut donc traiter les séquences multioctets comme des données opaques.

Le nombre maximal d'octets nécessaires pour stocker un caractère quelle que soit la localisation choisie sur le système est disponible dans la constante symbolique **MB_LEN_MAX** définie dans `<limits.h>`. De même, la variable **MB_CUR_MAX** – qui n'est pas une constante symbolique disponible lors de la compilation – indique le nombre maximal d'octets nécessaires pour stocker un caractère dans la localisation en cours.

Les routines que nous allons examiner ici sont déclarées dans `<stdlib.h>`. La fonction **wctomb()** – *wide char to multi-byte* – permet de convertir un caractère large en une séquence multioctet, alors que la fonction **mbtowc()** offre la conversion inverse :

```
int wctomb (char * destination, wchar_t source);
int mbtowc (wchar_t * destination, const char * source, size_t taille);
```

En fait, il ne faut jamais employer ces routines. Il en existe des équivalents avec la lettre `r` insérée avant le `to` et un argument supplémentaire en dernière position. Cet argument est de type `mbstate_t` et permet de mémoriser le *shift state* de la séquence multioctet. Cette valeur est un indicateur dépendant des caractères précédemment convertis. Elle n'est employée que dans certaines représentations multioctets, mais peut être indispensable. Il est nécessaire de la conserver lors de la manipulation successive des caractères d'une chaîne par exemple.

Pour initialiser un objet de type `mbstate_t`, il faut employer la fonction `memset()` ainsi :

```
    mbstate_t etat;
    memset (& etat, 0, sizeof (mbstate_t));
```

On pourra alors utiliser les routines **mbrtowc()** et **wcrtomb()**, qui permettent de faire les conversions :

```
size_t mbrtowc (wchar_t * destination,
                const char * source, size_t taille,
                mbstate_t * etat);
size_t wcrtomb (char * destination,
                wchar_t source,
                mbstate_t * etat);
```

La fonction `mbrtowc()` remplit le caractère large sur lequel on passe un pointeur en premier argument avec le résultat de la lecture de la séquence multioctet passée en second argument. On considère au maximum que le nombre d'octets indiqué en troisième position. Si la séquence est correcte, la fonction renvoie le nombre d'octets utilisés pour la conversion. Si la séquence débute bien mais que le nombre d'octets transmis est trop court, `mbrtowc()` renvoie –2, et si la séquence est définitivement invalide, elle renvoie –1. Si la conversion réussit, l'état transmis en dernier argument est mis à jour.

La routine `wcrtomb()` convertit le caractère large passé en second argument en séquence multioctet, qu'elle écrit dans la chaîne transmise en première position. Cette chaîne doit comporter au moins `MB_CUR_MAX` octets. La conversion n'a lieu que si le caractère large a une signification dans la localisation `LC_CTYPE` en cours.

Les fonctions **btowc()** et **wctob()** ne permettent de convertir qu'un seul octet en caractère large, et inversement. En fait, elles ne sont normalement utilisables que sur l'espace Ascii. Si le caractère large nécessite plusieurs octets pour être représenté, `wctob()` échoue en renvoyant EOF. Ces fonctions ne sont pas intéressantes, car elles obligent l'application à déterminer si un caractère large se représente sur un ou plusieurs octets, ce qui va à l'encontre des concepts d'internationalisation.

```
wint_t btowc (int caractere);
int wctob (wint_t caractere);
```

Pour calculer la longueur effective d'une séquence multioctet, on peut employer la fonction **mbrlen()**, qui permet d'examiner une chaîne dont la taille est indiquée en second argument et en renvoie la longueur jusqu'au caractère nul. Si la chaîne mentionnée est incomplète (la taille étant insuffisante pour obtenir un caractère multioctet entier), cette fonction renvoie –2.

```
size_t mbrlen (const char * chaine, size_t taille, mbstate_t * etat);
```

La fonction **mblen()** ne doit normalement pas être employée puisqu'elle ne peut pas mémoriser l'indicateur *shift state* de la chaîne.

```
int mblen (const char * chaine, size_t taille);
```

Les routines nommées **mbsrtowcs()** et **wcsrtombs()** convertissent des chaînes complètes en séquences multioctets et inversement, ainsi que **mbstowcs()** et **wcstombs()**, qui ne conservent pas l'état *shift state*.

```
size_t mbsrtowcs (wchar_t * destination,
                  const char ** chaine, size_t taille,
                  mbstate_t * etat);
size_t wcsrtombs (char * destination,
                  const wchar_t * *chaine, size_t taille,
                  mbstate_t * etat);
```

```
size_t mbstowcs  (wchar_t * destination,
                  const char * chaine, size_t taille);
size_t wcstombs  (char * destination,
                  const wchar_t * chaine, size_t taille);
```

Enfin, on peut noter l'existence des extensions Gnu **mbsnrtowcs()** et **wcsnrtombs()**, qui ne convertissent qu'une portion de la chaîne.

```
size_t mbsnrtowcs (wchar_t * destination,
                   const char ** chaine, size_t taille,
                   size_t maximum, mbstate_t * etat);
size_t wcsnrtombs (char * destination,
                   const wchar_t ** chaine, size_t taille,
                   size_t maximum, mbstate_t * etat);
```

Conclusion

Les utilisations des caractères larges ainsi que les conversions et échanges avec les séquences multioctets ne sont pas encore très répandus. Le support partiel de ces fonctionnalités par la bibliothèque C les rend encore un peu difficiles à employer dans des applications importantes. On peut toutefois prédire qu'il s'agira d'une évolution importante des programmes destinés à une diffusion internationale, et qu'il est donc bon de prévoir le plus tôt possible la compatibilité des applications avec ces standards. On pourra par exemple dans certains cas employer systématiquement des variables `wchar_t` et `wint_t` à la place de `char` et `int`, comme le font certaines portions de la GlibC.

24

Fonctions mathématiques

Linux dispose d'une panoplie de fonctions mathématiques couvrant l'essentiel des besoins courants. Il existe également des bibliothèques scientifiques supplémentaires permettant de répondre à des problèmes précis. On trouve d'ailleurs de nombreuses pages web consacrées aux logiciels scientifiques pour Linux.

Pour des besoins particuliers, on pourra donc compléter assez aisément les fonctions que nous allons décrire ici et qui sont définies par la GlibC.

Nous étudierons dans ce chapitre les fonctions trigonométriques, hyperboliques, exponentielles et logarithmiques. Nous verrons également des fonctions dont l'application est assez pointue, comme la fonction gamma ou les fonctions de Bessel. Nous examinerons ensuite les fonctions permettant de convertir un réel en entier, ainsi que le traitement des signes, les divisions entières et les modulo.

La plupart des fonctions mathématiques pouvant déclencher des erreurs, nous étudierons ici les moyens de les détecter, ainsi que le traitement des valeurs infinies. Ceci nous conduira d'ailleurs à analyser la méthode utilisée par la bibliothèque mathématique pour stocker les valeurs réelles.

Finalement, nous observerons un ensemble de générateurs aléatoires, ainsi que les « bonnes » manières de les utiliser.

L'essentiel des fonctions mathématiques est déclaré dans `<math.h>`. Lorsqu'on les utilise, il faut indiquer explicitement à l'éditeur de lien d'aller chercher les références nécessaires dans la bibliothèque `libm.so`. On ajoute donc l'option `-lm` sur la ligne de commande de `gcc`.

Sous Linux comme avec tout autre système d'ailleurs, il faut être très prudent lors des comparaisons de nombres réels. Le format utilisé pour stocker les valeurs réelles ne permet pas de disposer d'une précision absolue. Aussi, si un nombre peut être calculé de deux manières différentes, il est rare que les résultats coïncident, même s'ils sont mathématiquement égaux par définition. À titre d'exemple, nous savons que $\cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2}$

Nous allons calculer $\left[2 \times \cos\left(\frac{\pi}{4}\right)\right]^2 - 2$, en espérant retomber sur zéro.

exemple_math_1.c :

```
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    double d;
    d = cos(M_PI / 4) * 2.0;
    d = d * d - 2.0;
    fprintf(stdout, "(2 * cos(PI/4))² - 2 = %e \n", d);
    return EXIT_SUCCESS;
}
```

L'exécution suivante, vous vous en doutez probablement, ne donne pas le résultat escompté :

```
$ ./exemple_math_1
(2 * cos (PI/4))² - 2 = 2.734358e-16
$
```

Cette expérience démontre qu'il ne faut en aucun cas s'attendre à avoir des égalités parfaites avec les nombres réels manipulés sous forme numérique. Cela signifie que pour comparer deux nombres x et y, il ne faut pas utiliser simplement $x = y$ mais tester au contraire leur différence et vérifier si elle est suffisamment faible. La première approche est de considérer que x et y sont égaux si $|x - y| \leq \varepsilon$, avec ε petit. Toutefois, ceci n'est généralement pas très performant car la valeur choisie pour ε est figée, et un changement d'ordre de grandeur dans les unités utilisées pour x et y peut conduire à des résultats aberrants. Il est préférable, au prix d'une opération supplémentaire, de décider que x et y sont égaux si $|x - y| \leq \varepsilon(x + y)$. On peut dans ce cas fixer ε à une valeur assez faible devant 1 (par exemple 0,001), et la comparaison ne sera pas dépendante de l'ordre de grandeur de x et y.

Fonctions trigonométriques et assimilées

Toutes les fonctions trigonométriques courantes sont présentes dans la GlibC. Tous les angles considérés sont en radians. Lorsqu'on désire utiliser des valeurs en degrés pour l'interfaçage avec l'utilisateur, la conversion est aisée :

```
#define rad_2_deg(X)    (X / M_PI * 180.0)
#define deg_2_rad(X)    (X / 180.0 * M_PI)
```

La constante `M_PI` est définie par la GlibC dans `<math.h>`. Si, lors d'un portage de l'application, cette constante n'est pas définie, on peut la créer ainsi

```
#define M_PI 3.14159265358979323846264338327
```

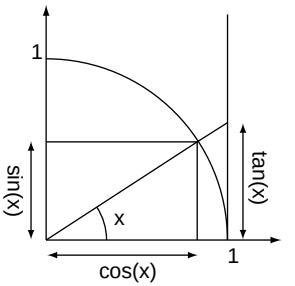
ou utiliser une variable globale initialisée au démarrage de l'application

```
    int    PI;
    int
main (void)
```

```
{
    PI = acos(-1.0);
    ...
}
```

Fonctions trigonométriques

Figure 24.1
Éléments trigonométriques usuels



- cosinus

```
double cos (double x);
float  cosf (float  x);
long double cosl (long double x);
```

Le cosinus de *x* est compris dans l'intervalle [–1,1].

$$\cos(0) = 1 \quad \cos\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \quad \cos\left(\frac{\pi}{2}\right) = 0 \quad \cos(\pi) = -1 \quad \cos\left(3\frac{\pi}{2}\right) = 0$$

- sinus

```
double sin (double x);
float  sinf (float x);
long double sinl (long double x);
```

Le sinus de *x* est compris dans l'intervalle [–1,1].

$$\sin(0) = 0 \quad \sin\left(\frac{\pi}{4}\right) = \frac{\sqrt{2}}{2} \quad \sin\left(\frac{\pi}{2}\right) = 1 \quad \sin(\pi) = 0 \quad \sin\left(3\frac{\pi}{2}\right) = -1$$

- tangente

```
double tan (double x);
float  tanf (float x);
long double tanl (long double x);
```

La tangente de *x* tend vers l'infini quand *x* tend vers $\frac{\pi}{2}, 3\frac{\pi}{2}, 5\frac{\pi}{2} \ldots$

Attention
Les routines `acos()`, `asin()` ou `tan()` par exemple peuvent échouer si leur argument n'est pas dans le domaine de définition de la fonction mathématique correspondante. Nous préciserons le moyen employé pour détecter les erreurs plus loin dans ce chapitre.

Fonctions trigonométriques inverses

- arc cosinus

```
double acos (double x);
float  acosf (float x);
long double acosl (long double x);
```

L'arc cosinus de x est l'angle compris dans $[0, \pi]$, dont le cosinus est x. L'argument x doit être obligatoirement dans $[-1, 1]$, sous peine de déclencher une erreur `EDOM`.

$$\arccos(-1) = \pi \quad \arccos(0) = \frac{\pi}{2} \quad \arccos(1) = 0$$

- arc sinus

```
double asin (double x);
float asinf (float x);
long double asinl (long double x);
```

L'arc sinus de x est l'angle dans l'intervalle $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, dont le sinus est égal à x. Ce dernier doit être obligatoirement dans l'intervalle $[-1, 1]$.

$$\arcsin(-1) = -\frac{\pi}{2} \quad \arcsin(0) = 0 \quad \arcsin(1) = \frac{\pi}{2}$$

- arc tangente

```
double atan (double x);
float  atanf (float x);
long double atanl (long double x);
```

L'arc tangente de x est l'angle compris dans $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, dont la tangente est égale à x. L'argument x peut prendre n'importe quelle valeur réelle. Plus x tend vers l'infini, plus son arc tangente tend vers $\frac{\pi}{2}$.

Fonctions connexes

- arc tangente complet

```
double atan2 (double x, double y);
float  atan2f (float x, float y);
long double atan2l (long double x, long double y);
```

Cette fonction calcule l'angle dont la tangente est égale à $\frac{y}{x}$. Pour cela, elle prend en compte le signe de chacune des deux variables afin de déterminer dans quel quadrant se trouve le résultat. L'angle renvoyé est situé dans $[-\pi, \pi]$. C'est typiquement la fonction qu'on doit utiliser lorsqu'on dispose du sinus et du cosinus d'un angle. Les arguments x et y ne sont pas obligatoirement dans l'intervalle $[-1, 1]$.

`atan2` (`cos` (x), `sin` (x)) = x

- hypoténuse

  Il existe une fonction nommée **hypot()**, très pratique pour les applications qui doivent mesurer des distances entre des points. Son prototype est le suivant :

```
double hypot (double x, double y);
float  hypotf (float x, float y);
long double hypotl (long double x, long double y);
```

  Elle renvoie la valeur $\sqrt{x^2 + y^2}$, calculée de manière optimisée. Ceci permet de disposer en une seule fonction de la distance entre deux points :

```
distance = hypot (point [i].x - point [j] . x,
                  point [i] . y - point [j] . y);
```

  Ce genre de calcul est très fréquent par exemple dans les routines de saisie de tracé vectoriel, où la position du clic de la souris est utilisée pour rechercher le polygone le plus proche et le sélectionner.

- sinus et cosinus

  Notons également la présence d'une extension Gnu nommée **sincos()**, qui permet de disposer en une seule fonction du sinus et du cosinus d'un angle :

```
void sincos (double angle, double * sinus, double * cosinus);
void sincosf (float angle, float * sinus,  float * cosinus);
void sincosl (long double angle, long double * sinus,
                                 long double * cosinus);
```

  On lui transmet bien entendu des pointeurs sur les variables qu'on désire remplir.

Fonctions hyperboliques

La bibliothèque C de Linux dispose des fonctions hyperboliques suivantes :

- cosinus hyperbolique

```
double cosh (double x);
float  coshf (float x);
long double coshl (long double x);
```

  Le cosinus hyperbolique de x est défini comme étant égal à $\frac{e^x + e^{-x}}{2}$.

- sinus hyperbolique

```
double sinh (double x);
float  sinhf (float x);
long double sinhl (long double x);
```

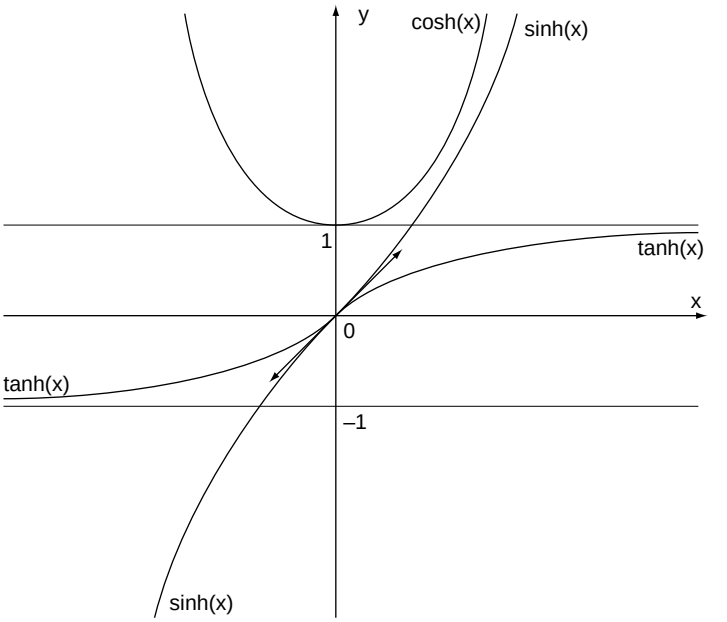
  Le sinus hyperbolique de x est défini comme étant égal à $\frac{e^x - e^{-x}}{2}$.

- tangente hyperbolique

```
double tanh(double x);
float  tanhf(float x);
long double tanhl(long double x);
```

  La tangente hyperbolique de x est définie comme étant égale à $\frac{\sinh(x)}{\cosh(x)}$.

Figure 24.2
Fonctions hyperboliques



- argument cosinus hyperbolique

```
double acosh (double x);
float  acoshf (float x);
long double acoshl (long double x);
```

L'argument cosinus hyperbolique de *x* est la valeur dont le cosinus hyperbolique est *x*. Ce dernier doit être supérieur ou égal à 1.

- argument sinus hyperbolique

```
double asinh (double x);
float  asinhf (float x);
long double asinhl (long double x);
```

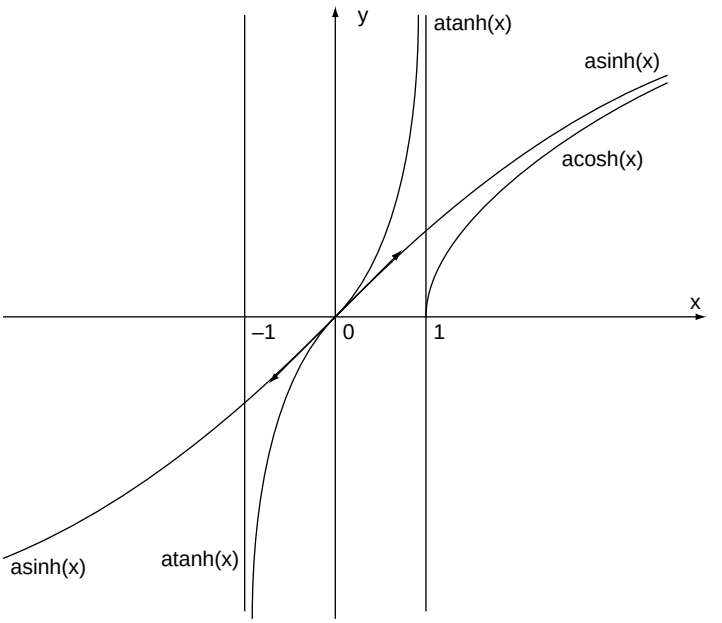
L'argument sinus hyperbolique de *x* est la valeur dont le sinus hyperbolique est *x*.

- argument tangente hyperbolique

```
double atanh (double x);
float  atanhf (float x);
long double atanhl (long double x);
```

L'argument tangente hyperbolique de *x* est la valeur dont la tangente hyperbolique est *x*. La valeur absolue de ce dernier doit être inférieure à 1. Si elle est égale à 1, l'argument tangente hyperbolique est infini.

Figure 24.3
Fonctions arguments hyperboliques



Exponentielles, logarithmes, puissances et racines

Fonctions exponentielles

- exponentielle

```
double exp (double x);
float  expf (float x);
long double expl (long double x);
```

Cette fonction renvoie e^x, e étant le nombre de base des logarithmes népériens 2,7182818285…

- exponentielle moins 1

```
double expm1 (double x);
float  expm1f (float x);
long double expm1l (long double x);
```

Cette fonction renvoie $e^x - 1$. Le calcul est effectué en gardant un maximum de précision, même lorsque x tend vers 0 (donc e^x vers 1).

- exponentielle en base 2

```
double exp2 (double x);
float  exp2f (float x);
long double exp2l (long double x);
```

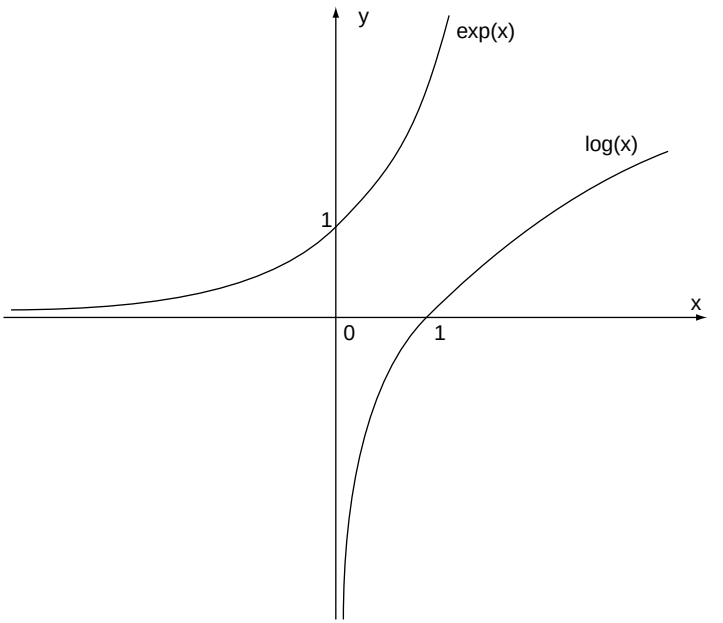
Cette fonction calcule 2^x, qui est équivalent à $e^{x\ln(2)}$.

- exponentielle en base 10

```
double exp10 (double x);
float  exp10f (float x);
long double exp10l (long double x);
```

Cette fonction calcule 10^x.

Figure 24.4
Fonctions exponentielles et logarithmes



Fonctions logarithmiques

- logarithme népérien

```
double log (double x);
float  logf (float x);
long double logl (long double x);
```

Cette fonction renvoie le logarithme népérien (naturel) de x, c'est-à-dire la valeur y pour laquelle $e^y = x$. L'argument x doit être strictement positif.

- logarithme népérien de 1 plus x

```
double log1p (double x);
float  log1pf (float x);
long double log1pl (long double x);
```

Cette fonction calcule `log(1 + x)` en gardant le maximum de précision, même lorsque x tend vers zéro, dans ce cas `log(1 + x)` tend aussi vers zéro. Alors x doit être strictement supérieur à –1.

- logarithme en base 2

```
double log2p (double x);
float  log2pf (float x);
long double log2pl (long double x);
```

Cette fonction calcule le logarithme en base 2 de *x*. Cette valeur est souvent utilisée pour connaître le nombre minimal de bits nécessaires pour coder un nombre.

- logarithme décimal

```
double log10 (double x);
float  log10 (float x);
long double log10 (long double x);
```

Cette fonction calcule le logarithme en base 10 de *x*. Ceci permet de connaître le nombre de chiffres décimaux nécessaires pour afficher la partie entière de *x*.

Puissances et racines

- élévation à la puissance

```
double pow (double x, double y);
float  powf (float x, float y);
long double powl (long double x, long double y);
```

Cette fonction renvoie x^y. Si *x* est négatif et si *y* n'est pas un entier, x^y devrait être complexe. Dans ce cas, la fonction échoue et renvoie une erreur `EDOM`.

- racine carrée

```
double sqrt (double x);
float  sqrtf (float x);
long double sqrtl (long double x);
```

La fonction `sqrt()` renvoie la racine carrée (*square root*) de *x*. Bien entendu, *x* doit être positif ou nul.

- racine cubique

```
double cbrt (double x);
float  cbrtf (float x);
long double cbrtl (long double x);
```

Cette fonction renvoie la racine cubique (*cube root*) de *x*. Il n'y a pas de contraintes sur les valeurs de *x*.

Nombres complexes

Depuis la norme C9X, un nouveau type de données est disponible pour représenter les nombres complexes. Il faut pour cela inclure l'en-tête `<complex.h>`. Nous pouvons dès lors définir des variables de type `float complex`, `double complex`, ou `long double complex` en fonction de l'étendue désirée, et utiliser des fonctions spécifiques pour les manipuler.

Nous disposons de la constante symbolique `I` pour représenter le nombre imaginaire pur *i* tel que $i^2 = -1$. Voici un exemple simple dans lequel nous initialisons un nombre complexe, puis

affichons ses divers paramètres, calculons son conjugué et affichons les caractéristiques de celui-ci :

exemple_complexe :

```
        #include <complex.h>
        #include <math.h>
        #include <stdio.h>
        #include <stdlib.h>

        int
main (void)
{
        double complex z;

        z = 0.5  + I * (sqrt(3)/2);

        fprintf(stdout, "Z : \n");
        fprintf(stdout, "  Partie réelle     : %f\n", creal(z));
        fprintf(stdout, "  Partie imaginaire : %f\n", cimag(z));
        fprintf(stdout, "  Module            : %f\n", cabs(z));
        fprintf(stdout, "  Argument          : %f\n", carg(z));

        z = conj(z);

        fprintf(stdout, "\nConjugué de Z : \n");
        fprintf(stdout, "  Partie réelle     : %f\n", creal(z));
        fprintf(stdout, "  Partie imaginaire : %f\n", cimag(z));
        fprintf(stdout, "  Module            : %f\n", cabs(z));
        fprintf(stdout, "  Argument          : %f\n", carg(z));

        return EXIT_SUCCESS;
}
```

L'exécution donne :

```
$ ./exemple_complexe
Z :
  Partie réelle     : 0.500000
  Partie imaginaire : 0.866025
  Module            : 1.000000
  Argument          : 1.047198

Conjugué de Z :
  Partie réelle     : 0.500000
  Partie imaginaire : -0.866025
  Module            : 1.000000
  Argument          : -1.047198
$
```

La valeur 0.866025 correspond à $\frac{\sqrt{3}}{2}$ et 1.047198 à $\frac{\Pi}{3}$.

Toutes les routines que nous avons vues précédemment sont disponibles dans une forme complexe. Les prototypes suivants correspondent aux versions `double complex` mais il existe également des fonctions `float complex` (avec un suffixe `f`) et `long double complex` (avec un suffixe `l`).

```
double complex ccos (double complex z);
double complex csin (double complex z);
double complex ctan (double complex z);
double complex cacos (double complex z);
double complex casin (double complex z);
double complex catan (double complex z);
double complex ccosh (double complex z);
double complex csinh (double complex z);
double complex ctanh (double complex z);
double complex cacosh (double complex z);
double complex casinh (double complex z);
double complex catanh (double complex z);
double complex cexp (double complex z);
double complex clog (double complex z);
double complex cpow (double complex z1, double complex z2);
double complex csqrt (double complex z);
```

Le lecteur désirant plus de détails sur l'une de ces fonctions pourra se reporter aux pages de manuel, ou à la description dans la norme SUSv3.

Calculs divers

La bibliothèque GlibC offre quelques fonctions qui ne s'appliquent que dans des cas très particuliers et assez rares.

Fonctions d'erreur

- erreur

```
double erf (double x);
float  erff (float x);
long double erfl (long double x);
```

La fonction d'erreur de *x*, Erf`(x)`, est utilisée dans le domaine du calcul des probabilités. Elle a été définie par Gauss ainsi :

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Cette fonction tend très vite vers 1, par valeur inférieure lorsque *x* tend vers +∞.

- erreur complémentaire

```
double erfc (double x)
float erfcf (float x)
long double erfcl (long double x)
```

renvoie l'erreur complémentaire définie ainsi : `erfc(x)` = 1 – `erf(x)`. Le calcul est réalisé en conservant la précision, même lorsque *x* est grand.

Fonction gamma

- gamma

```
double tgamma (double x);
float  tgammaf (float x);
long double tgammal (long double x);
```

tgamma (x) = $\Gamma(x)$

La fonction gamma est aussi appelée fonction eulérienne de deuxième espèce.

Sa définition est la suivante :

$$\Gamma(x) = \int_0^{\infty} e^{-t} t^{x-1} dt$$

Cette fonction a une propriété importante : si n est un entier naturel, alors $\Gamma(n + 1) = n!$. La fonction gamma est donc une extrapolation de la factorielle sur l'ensemble des réels.

- logarithme de gamma

```
double lgamma (double x);
float  lgammaf (float x);
long double lgammal (long double x);
```

lgamma (x) = log ($\Gamma(x)$)

Fonctions de Bessel

- Bessel de première espèce

```
double j0 (double x);
float  j0f (float x);
long double j0l (long double x);

double j1 (double x);
float  j1f (float x);
long double j1l (long double x);

double jn (int n, double x);
float  jnf (int n, float x);
long double jnl (int n, long double x);
```

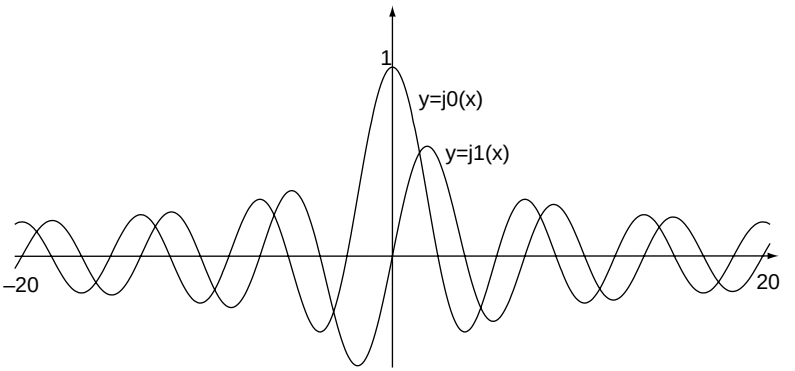
Ces trois fonctions sont appelées fonctions de Bessel de première espèce, respectivement d'ordre 0, 1, et n. La définition d'une fonction de Bessel d'ordre n est la suivante :

$$J_n(x) = \left(\frac{x}{2}\right)^n \sum_{k=0}^{\infty} \frac{(-1)^k}{k!\,\Gamma(n+k+1)} \left(\frac{x}{2}\right)^{2k}$$

On retrouve la fonction gamma, vue plus haut, dans l'expression des fonctions de Bessel.

Les fonctions de Bessel sont appliquées dans des domaines assez divers, tels que l'électromagnétisme, la thermodynamique, l'acoustique…

Figure 24.5
Fonctions de Bessel de première espèce

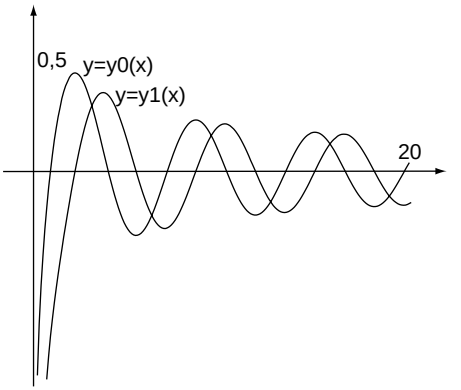


- Bessel de seconde espèce

```
double y0 (double x);
float  y0f (float x);
long double y0l (long double x);

double y1 (double x);
float  y1f (float x);
long double y1l (long double x);

double yn (int n, double x);
float  ynf (int n, float x);
long double ynl (int n, long double x);
```

Les fonctions de Bessel de seconde espèce, d'ordre 0, 1, et *n*, sont moins utilisées que celles de première espèce.

Figure 24.6
Fonctions de Bessel de seconde espèce

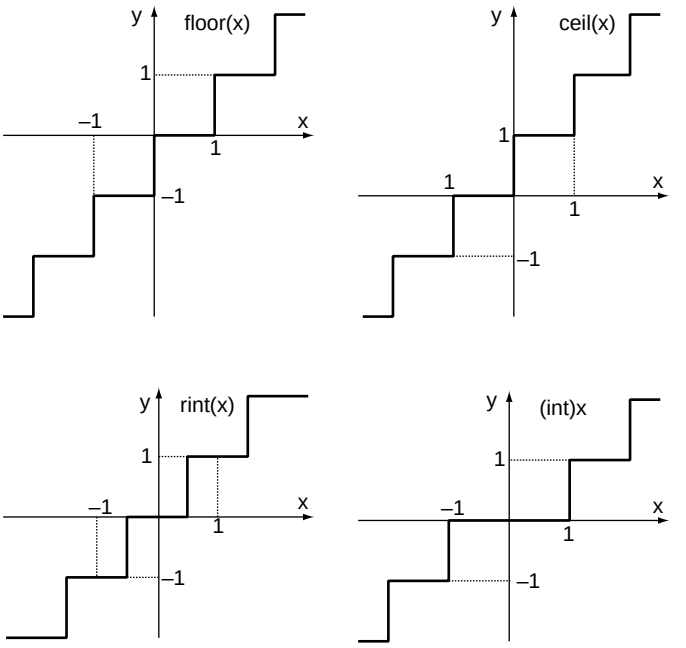


Limites d'intervalles

Il est fréquent de devoir transformer le résultat de calculs réels en valeurs entières. Toutefois, il y a plusieurs fonctions disponibles, et un mauvais choix peut conduire à des erreurs de conversion assez déroutantes.

La conversion la plus simple est celle qui est implicite lorsqu'on transfère le contenu d'une variable réelle dans une variable entière. Cette conversion consiste simplement à supprimer la partie décimale du nombre réel. Ainsi 4,5 devient 4, et –3,2 devient –3. Fréquemment employée, cette méthode est pourtant rarement celle qui est voulue en pratique. Dans un affichage cartographique par exemple, les polygones sont généralement représentés par des listes de points dont les coordonnées sont réelles. Ainsi, il est possible de réaliser des opérations de zoom, translation ou rotation avec une bonne précision. Toutefois, lors de l'affichage, une conversion en valeurs entières doit avoir lieu pour obtenir les coordonnées des pixels. Si on utilise une conversion implicite des variables du langage C, on risque de voir deux polygones adjacents mal raccordés par leurs sommets communs.

Figure 24.7
Fonctions d'arrondi



Pour éviter ce problème, on emploie plutôt la fonction `rint()` qui arrondit à l'entier le plus proche :

```
double rint (double x);
float  rintf (float x);
long double rintl (long double x);
```

On observe alors que `rint(4.8)=5`, `rint(4.2)=4`, `rint(-3.1)=-3` et `rint(-0.9)=-1`

Parfois, on peut également préférer utiliser la véritable fonction mathématique « partie entière », qui arrondit à l'entier immédiatement inférieur ou égal. Cette fonction est nommée **`floor()`** :

```
double floor (double x);
float  floorf (float x);
long double floorl (long double x);
```

Cette fois, `floor(1.2)=1`, et `floor(1.9)=1`, mais également `floor(-0.9)=-1`. On notera qu'il existe un synonyme de `floor()` nommé **`trunc()`**.

Enfin, la fonction **`ceil()`** arrondit symétriquement par excès à l'entier immédiatement supérieur :

```
double ceil (double x);
float  ceilf (float x);
long double ceill (long double x);
```

Voyons les différences de comportement de ces quatre routines autour de zéro.

exemple_math_2.c :

```
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    double d;
    double arrondi_inf;
    double arrondi_sup;
    double arrondi_proche;
    int    converti;

    printf("réel    floor() ceil()  rint()  (int)\n");
    for (d = -1.8; d < 1.9; d += 0.2) {
        arrondi_inf = floor(d);
        arrondi_sup = ceil(d);
        arrondi_proche = rint(d);
        converti = (int) d;
        printf("% 4.1f    % 4.1f    % 4.1f    % 4.1f    % 2d\n",
            d, arrondi_inf, arrondi_sup, arrondi_proche, converti);
    }

    return EXIT_SUCCESS;
}
```

L'exécution correspond à ce qu'on attendait :

```
$ ./exemple_math_2
réel    floor() ceil()  rint()  (int)
-1.8    -2.0    -1.0    -2.0    -1
-1.6    -2.0    -1.0    -2.0    -1
```

```
-1.4    -2.0    -1.0    -1.0    -1
-1.2    -2.0    -1.0    -1.0    -1
-1.0    -1.0    -0.0    -1.0     0
-0.8    -1.0    -0.0    -1.0     0
-0.6    -1.0    -0.0    -1.0     0
-0.4    -1.0    -0.0    -0.0     0
-0.2    -1.0    -0.0    -0.0     0
 0.0     0.0     1.0     0.0     0
 0.2     0.0     1.0     0.0     0
 0.4     0.0     1.0     0.0     0
 0.6     0.0     1.0     1.0     0
 0.8     0.0     1.0     1.0     0
 1.0     1.0     2.0     1.0     1
 1.2     1.0     2.0     1.0     1
 1.4     1.0     2.0     1.0     1
 1.6     1.0     2.0     2.0     1
 1.8     1.0     2.0     2.0     1
$
```

Valeurs absolues et signes

Il existe plusieurs fonctions permettant d'extraire la valeur absolue d'un nombre, suivant le type de donnée utilisée.

```
int    abs  (int n);
long   labs (long n);
long long llabs (long long n);

double fabs (double x);
float fabsf (float x);
long double fabsl (long double x);
```

Les trois premières fonctions travaillent avec des entiers, les trois dernières avec des réels. Pour éviter qu'un programme, qui ne réalise que des opérations arithmétiques sur des entiers, ne soit obligé d'inclure `<math.h>` et toute la surcharge de code d'émulation mathématique sur certains systèmes, les fonctions `abs()`, `labs()` et `llabs()` sont déclarées dans `<stdlib.h>` depuis la norme C9X.

La représentation d'un entier sur *n* bits permet de couvrir l'intervalle allant de -2^n à $2^n - 1$. Aussi, il n'est pas possible avec `abs()` ou `labs()` de calculer la valeur absolue du plus petit entier représentable dans le type de données correspondant. En effet, le débordement nous ramène à la même valeur négative.

Il existe une fonction nommée **`copysign()`** permettant d'extraire le signe d'un nombre réel de manière efficace.

```
double copysign (double valeur, double signe);
float copysignf (float valeur, float signe);
long double copysignl (long double valeur, long double signe);
```

Cette fonction renvoie un nombre constitué de la valeur absolue de son premier argument et du signe du second. Cette fonction est utilisable avec les infinis.

Divisions entières, fractions, modulo

Il existe plusieurs fonctions permettant de calculer des divisions entières. Rappelons que les opérateurs « / » et « % » du langage C permettent aussi de calculer facilement un quotient et un reste.

- division entière

```
div_t div (int dividende, int diviseur);
```

Cette fonction effectue la division entière $\frac{dividende}{diviseur}$ et renvoie le résultat dans une structure disposant des membres suivants :

Nom	Type	Signification
`quot`	`Int`	Quotient de la division entière
`rem`	`Int`	Reste de la division entière

```
ldiv_t ldiv (long dividende, long diviseur);
lldiv_t lldiv (long long dividende, long long diviseur);
```

Les fonctions `ldiv()` et `lldiv()` sont calquées sur `div()`, simplement elles renvoient leurs résultats dans des structures `ldiv_t` et `lldiv_t`, dont les membres (également nommés `quot` et `rem`) sont de type `long` pour l'une et `long long` pour l'autre.

- modulo

```
double fmod (double dividende, double diviseur);
float fmodf (float dividende, float diviseur);
double fmodl (long double dividende, long double diviseur);

double drem (double dividende, double diviseur);
```

Toutes ces fonctions permettent de calculer le reste d'une division entière mais avec des définitions différentes. Les fonctions `fmod()` renvoient un nombre dont le signe est celui du dividende et dont la valeur absolue est dans l'intervalle [0, *diviseur*], alors que `drem()` qui est une extension BSD non définie par SUSv3, fournit un résultat dans $\left[-\frac{diviseur}{2}, \frac{diviseur}{2}\right]$.

En fait, toutes renvoient (*dividende* – $n \times$ *diviseur)*, simplement `fmod()` arrondit *n* systématiquement à l'entier inférieur, alors que `drem()` l'arrondit à l'entier le plus proche.

```
double modf (double valeur, double * partie_entiere);
float modff (float valeur, float * partie_entiere);
long double modfl (long double valeur, long double * partie_entiere);
```

Cette fonction sépare la partie décimale et la partie entière de son premier argument. Elle renvoie la partie décimale après avoir rempli le pointeur passé en second argument avec la partie entière. Par exemple avec :

```
double partie_decimale, partie_entiere;
partie_decimale = modf (7.67, & partie_entiere);
partie_decimale vaudra 0.67 et partie_entiere 7.
```

La partie entière est calculée en utilisant la conversion implicite de réel en entier, aussi pour les valeurs négatives, la partie décimale se trouve dans l'intervalle]–1, 0].

- fraction normalisée

```
double frexp (double valeur, double exposant);
float frexpf (float valeur, float exposant);
long double frexpl (long double valeur, long double exposant);
```

Cette fonction sert à décomposer un nombre en virgule flottante en une fraction normalisée, se trouvant dans l'intervalle $\left[\frac{1}{2}, 1\right[$, et un exposant. Lorsqu'on multiplie cette fraction normalisée par 2^{exposant}, on retrouve la valeur originale. Cette fonction est en fait l'inverse de `ldexp()` présentée ci-dessous. Nous examinerons dans la prochaine section le stockage des réels en mémoire, ce qui éclairera un peu l'utilité de cette fonction.

```
double ldexp (double x, double y);
float ldexpf (float x, float y);
long double ldexpl (long double x, long double y);
```

Cette fonction renvoie la valeur $x \cdot 2^y$. Ceci sert pour reconstituer un nombre réel à partir de sa représentation binaire au format IEEE 754.

Infinis et erreurs

Les fonctions mathématiques ont des domaines de définition bien précis. Essayer d'invoquer une fonction, par exemple `log()`, pour une valeur interdite (disons -5) doit renvoyer une erreur. Toutefois, la routine `log()` ne peut pas se contenter de renvoyer –1 en cas d'erreur, comme le font d'autres fonctions de bibliothèque habituellement. Cette valeur, en effet, est tout à fait légitime pour $x = \frac{1}{e}$.

Valeur non numérique

Pour signaler une erreur, les routines renvoient une valeur spéciale, nommée **`NaN`**, ce qui signifie *Not a Number*. De plus, elles positionnent la variable globale `errno` (avec l'erreur `EDOM` en général). Pour vérifier le résultat, il existe une fonction nommée **`isnan()`**, déclarée ainsi :

```
int isnan (double valeur);
```

Il s'agit en réalité d'une macro, aussi peut-on utiliser la même routine pour les différents types de réels (`float`, `double` et `long double`).

Elle renvoie 0 si son argument est numérique et une valeur non nulle sinon. On peut donc employer le code suivant :

```
    double cosinus ;
    double angle :

    angle = acos(cosinus) ;
    if (isnan(angle)) {
        perror("acos");
```

```
        exit(EXIT_FAILURE);
    }
    ...
```

Attention

Il n'existe pas de constante symbolique `NaN` avec laquelle on pourrait faire la comparaison. Nous verrons dans la représentation binaire des réels qu'il n'y a pas une unique valeur non numérique, mais qu'on en trouve une multitude.

Infinis

Parfois une fonction réelle, par exemple $f(x) = \frac{1}{x}$, peut renvoyer une valeur infinie sur un point précis de son intervalle de définition, en $x = 0$ en l'occurrence. Comme la précision de la représentation des réels en virgule flottante est limitée, il existe nécessairement une certaine zone de « flou » autour de ce point. On ne peut donc pas simplement renvoyer une erreur, mais la bibliothèque C doit permettre de traiter les infinis. Elle utilise donc deux valeurs supplémentaires spéciales, indiquant $+\infty$ et $-\infty$. Pour les détecter, il existe une fonction **`isinf()`** :

```
int isinf (double valeur);
```

Cette routine renvoie 0 si la valeur est finie, –1 s'il s'agit de $-\infty$, et + 1 s'il s'agit de $+\infty$.

Il existe également une routine **`finite()`** ayant le fonctionnement contraire :

```
int finite (double valeur);
```

Elle renvoie une valeur non nulle si la valeur transmise est numérique (pas `NaN`) et finie. Le paramètre indiqué ici est de type `double` mais on utilise les mêmes routines `isinf()` et `finite()` quel que soit le type de données réelles.

Voici un exemple qui va nous permettre de voir les divers cas traités par la bibliothèque mathématique.

exemple_math_3.c :

```
    #define _XOPEN_SOURCE 600
    #include <errno.h>
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    void
affiche_nombre (const char * chaine, double d)
{
    fprintf(stdout, "%s", chaine);
    if (isnan(d))
        fprintf(stdout, "Indéfini \n");
    else if (isinf(d) == 1)
        fprintf(stdout, "+ Infini \n");
```

```
    else if (isinf(d) == -1)
        fprintf(stdout, "- Infini \n");
    else
        fprintf(stdout, "%f \n", d);
}

    int
main (void)
{
    double  d;

    d = +0.0;
    d = 1.0 / d;
    affiche_nombre("1 / +0 = ", d);
    d = -0.0;
    d = 1.0 / d;
    affiche_nombre("1 / -0 = ", d);
    d = 0.0 / 0.0;
    affiche_nombre("0 / 0 = ", d);
    d = log(0.0);
    affiche_nombre("log(0) = ", d);
    d = log(-1.0);
    affiche_nombre("log(-1)= ", d);
    d = MAXFLOAT;
    affiche_nombre("MAXFLOAT = ", d);
    d = exp(MAXFLOAT);
    affiche_nombre("exp(MAXFLOAT)= ", d);
    return EXIT_SUCCESS;
}
```

Pour des raisons de compatibilité avec les standards précédents, il faut définir la constante `_XOPEN_SOURCE` avec la valeur `600` pour obtenir `MAXFLOAT` et `isinf()`.

Nous remarquons que la bibliothèque distingue +0 de –0, ce qui peut paraître surprenant à première vue, mais qui s'explique par la représentation des nombres que nous examinerons plus bas.

```
$ ./exemple_math_3
1 / +0 = + Infini
1 / -0 = - Infini
0 / 0 = Indéfini
log (0) = - Infini
log (-1)= Indéfini
MAXFLOAT = 340282346638528859811704183484516925440.000000
exp(MAXFLOAT)= + Infini
$
```

Si on construit une bibliothèque mathématique complémentaire, il peut être nécessaire de renvoyer des valeurs infinies ou non numériques en cas d'erreur. Il ne serait pas très élégant d'être obligé de les obtenir avec des artifices du genre `(-1.0 / 0.0)` ou `log (-1.0)`. Il faut donc avoir un moyen d'accéder directement à ces valeurs. Les infinis sont représentés par `HUGE_VAL` ou –`HUGE_VAL`. Il n'y a pas de constante symbolique permettant de renvoyer directement

NaN, mais il existe une routine BSD non définie par SUSv3 pouvant servir à renvoyer une erreur.

```
double infnan (int erreur);
```

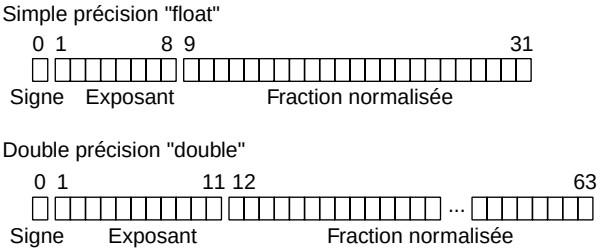
Si l'argument erreur vaut EDOM, infnan() renvoie NaN, s'il vaut ERANGE ou –ERANGE, infnan() renvoie respectivement HUGE_VAL ou –HUGE_VAL.

Représentation des réels en virgule flottante

Pour stocker un réel en mémoire, la plupart des ordinateurs utilisent un format normalisé, connu sous le nom de IEEE 754. Ce format peut donc convenir pour transférer des valeurs numériques entre ordinateurs. Toutefois, il est important de savoir éventuellement décoder « manuellement » les données si le système destinataire ne respecte pas le même format.

Le format IEEE 754 utilise 32 bits pour les valeurs de type float, 64 bits pour les réels de type double. Dans les deux cas, le réel est stocké sous forme d'un bit de signe, suivi d'un exposant (sur 8 bits dans un cas, et 11 bits dans l'autre), suivi de la fraction normalisée sur 23 et 52 bits respectivement.

Figure 24.8
Représentations binaires des nombres réels



Le format long double est défini par IEEE 854. Il s'agit d'un codage sur 12 octets, soit 96 bits, composés d'un bit de signe, suivi par 15 bits d'exposant, 16 bits vides, puis 64 bits de fraction normalisée.

Le bit de signe vaut 0 si le nombre est positif, et 1 s'il est négatif. Ceci nous explique pourquoi la bibliothèque distingue +0.0 et –0.0.

L'exposant est compris entre 0 et 255 pour les float, 2 047 pour les réels de type double, ou 32 767 pour les long double.

Si l'exposant vaut 255 (2 047 ou 32 727) et si la fraction normalisée n'est pas nulle, alors le nombre représente NaN. Si l'exposant vaut 255 (2 047 ou 32 767) et si la fraction est nulle, le nombre correspond à $+\infty$ ou $-\infty$ en fonction du bit de signe. On comprend alors qu'il existe une grande quantité de valeurs pouvant correspondre à NaN, puisqu'il suffit que la fraction normalisée ne soit pas nulle. C'est pour cela qu'il n'existe pas de constante symbolique NaN avec laquelle on pourrait comparer une valeur.

Si l'exposant n'est pas nul et s'il n'a pas sa valeur maximale (255, 2 047 ou 32 767), alors le nombre représenté vaut :

$(-1)^{signe} \times 2^{(exposant - 127)} \times (1.0 + fraction)$ pour les réels de type float,

$(-1)^{signe} \times 2^{(exposant - 1\,023)} \times (1.0 + fraction)$ pour les réels de type double, et

$(-1)^{signe} \times 2^{(exposant - 16\,383)} \times (1.0 + fraction)$ pour les long double.

La valeur de la fraction est calculée en additionnant les puissances négatives successives de 2, en commençant par $\frac{1}{2}$ et en se terminant par $\frac{1}{2^{23}} = \frac{1}{8388608}, \frac{1}{2^{52}}$ ou $\frac{1}{2^{64}}$.

Par exemple, la fraction normalisée 11010010 (en se limitant à 8 bits) vaut :

$$\frac{1}{2} + \frac{1}{4} + 0 + \frac{1}{16} + 0 + 0 + \frac{1}{128} + 0 = 0.8203125 .$$

Finalement un cas particulier se présente si l'exposant vaut zéro.

Si la fraction est nulle, le nombre correspond à +0 ou –0 en fonction du bit de signe, sinon, la valeur est :

$(-1)^{signe} \times 2^{-126} \times fraction$ pour un `float`,

$(-1)^{signe} \times 2^{-1\,022} \times fraction$ pour un `double`, et

$(-1)^{signe} \times 2^{-16\,382} \times fraction$ pour un `long double`.

Si nous avons présenté ce format ici, c'est qu'il est largement utilisé dans les ordinateurs modernes et permet normalement un échange assez facile des données. On peut ainsi sur 4, 8, ou même 12 octets, transférer des valeurs réelles avec une très bonne précision sur un réseau ou dans un fichier. La connaissance détaillée des formats IEEE 754 et 854 nous permet de conserver dans un tiroir des routines d'encodage et de décodage s'il faut porter une application sur un système représentant différemment les nombres en virgule flottante.

Générateurs aléatoires

Il existe trois types de générateurs aléatoires disponibles sous Linux. L'un est offert par le noyau, le deuxième par la bibliothèque C standard, et le troisième par la bibliothèque mathématique. Chacun présente des avantages et des inconvénients.

Générateur aléatoire du noyau

Linux offre un générateur aléatoire intégré, sous forme de deux fichiers spéciaux de périphérique, `/dev/random` et `/dev/urandom`. Ils doivent être créés avec les numéros majeurs 1 et mineurs 8 et 9 respectivement :

```
$ ls -l /dev/*random
crw-r--r--   1 root     root       1,   8 May  5  1998 /dev/random
crw-r--r--   1 root     root       1,   9 May  5  1998 /dev/urandom
$
```

Les caractères qu'on trouve dans ces pseudo-fichiers sont engendrés à partir de sources de bruit définies dans les pilotes de périphériques. Le noyau extrait des données aléatoires à partir de mesures diverses imprévisibles. Ces caractères sont disponibles dans le fichier `/dev/random`. Lorsque le système n'a plus de données assez bruitées à sa disposition, l'appel système de lecture sera bloquant. Il est assez amusant, sur un système au repos, de demander un `cat < /dev/random` et d'observer que l'affichage s'arrête au bout d'un moment, et que le noyau retrouve à nouveau des valeurs aléatoires à chaque déplacement de la souris ou action sur le clavier. Le fait que l'appel système soit bloquant si des données vraiment aléatoires ne sont plus disponibles peut être parfaitement adapté dans certains cas (création de mots de

passe, clé cryptographique[1]…) mais très gênant dans d'autres situations (jeux). Pour cela, le noyau offre également un autre pseudo-fichier, `/dev/urandom`, dont la lecture n'est jamais bloquante, mais dont les valeurs peuvent devenir moins aléatoires lorsqu'il n'y a plus assez de données bruitées. Le noyau emploie alors un algorithme déterministe, et les caractères obtenus peuvent théoriquement être devinés à l'avance ; il est donc à éviter dans les applications cryptographiques.

La lecture depuis ces pseudo-fichiers fournit probablement la meilleure moisson de valeurs aléatoires, puisque cette méthode est la plus proche du générateur idéal qui consisterait à numériser un bruit blanc parfait, fournissant ainsi des données totalement imprévisibles.

Toutefois, cette méthode est difficilement portable sur des systèmes moins accommodants que Linux, aussi est-on parfois obligé de se rabattre sur des générateurs purement numériques.

Générateur aléatoire de la bibliothèque C standard

Un générateur numérique ne peut fournir que des valeurs pseudo-aléatoires. Cela signifie que la série de nombres fournie se répétera nécessairement, mais avec une période tellement longue que l'observation externe de la séquence, sur un échantillon de taille raisonnable, ne permettra pas de prédire la valeur suivante. En général, les générateurs doivent être initialisés avec une valeur qui sert de racine pour engendrer la série aléatoire. Si on réinitialise le générateur avec la même racine, il redonnera une séquence identique. Ceci est particulièrement précieux pour le débogage d'une application. Par contre, pour rendre la séquence imprévisible, il faut utiliser une racine elle-même la plus aléatoire possible (par exemple en lisant `/dev/random`).

Commençons tout d'abord par les fonctions **`rand()`** et **`srand()`** qui sont définies par la norme C Ansi et déclarées dans `<stdlib.h>` :

```
int rand (void);
```

Cette fonction renvoie un nombre pseudo-aléatoire d'une série uniformément répartie dans l'intervalle [0, `RAND_MAX`]. La constante symbolique `RAND_MAX` correspond au plus grand nombre aléatoire disponible. Le générateur employé par la GlibC fournit des nombres dont les bits de poids faibles sont aussi aléatoires que les bits de poids forts[2]. On peut donc utiliser n'importe quelle méthode pour réduire l'intervalle [0, `RAND_MAX`] à la plage de valeurs désirées (en prenant garde à pouvoir atteindre correctement les extrémités).

```
void srand (unsigned int racine);
```

Cette fonction permet d'initialiser la séquence de nombres pseudo-aléatoires de `rand()`. Lorsqu'on réinitialise la séquence avec la même valeur, on obtient les mêmes nombres pseudo-aléatoires. Par défaut, la séquence est initialisée à 1 si on invoque `rand()` avant de fournir une racine. Il y a plusieurs méthodes permettant de choisir une racine correcte. La plus simple, dans le cas d'un jeu par exemple, consiste à utiliser la date et l'heure, exprimées en secondes écoulées depuis le 1er janvier 1970 :

```
srand (time (NULL));
```

1. Il ne faut pas oublier que ces fichiers peuvent être falsifiés par *root* (tout comme la bibliothèque C d'ailleurs), et il ne faut pas leur accorder une confiance aveugle en terme de sécurité.
2. Ce n'était pas le cas dans l'implémentation traditionnelle de `rand()` sous Unix, aussi devait-on prendre garde à employer de préférence les bits de poids forts qui étaient moins prévisibles que ceux de poids faibles.

Ainsi, la séquence sera différente à chaque lancement de l'application. Par contre, le comportement est prévisible. Pour engendrer des mots de passe, on préférera utiliser une racine provenant d'un générateur aléatoire externe, comme `/dev/random`.

La valeur courante de la séquence est mémorisée dans une variable globale. Si on veut pouvoir accéder à une séquence répétable (pour le débogage) dans une application multithread, on peut utiliser l'extension **`rand_r()`** :

```
int rand_r (unsigned int * racine) ;
```

Elle renvoie une valeur aléatoire et stocke dans le pointeur fourni en argument son état actuel. Le problème est que le type `unsigned int` est rarement assez grand pour permettre l'implémentation d'un bon générateur aléatoire. On adoptera de préférence dans ce cas les extensions Gnu décrites dans la prochaine section.

Il existe des fonctions BSD déclarées dans `<stdlib.h>`, **`random()`**, **`srandom()`**, **`initstate()`** et **`setstate()`**, qui ont à peu près les mêmes fonctionnalités. Ces routines sont à présent considérées comme obsolètes car elles sont limitées à des entiers sur 32 bits :

```
int random (void) ;
void srandom (unsigned int racine) ;
```

Ces routines représentent le pendant de `rand()` et `srand()`. Les fonctions `initstate()` et `setstate()` ont une interface compliquée et servent simplement à sauvegarder ou à restituer l'état du générateur en utilisant un tableau d'entiers. Elles ne sont normalement plus utilisées.

Générateur aléatoire de la bibliothèque mathématique

Le générateur aléatoire de la bibliothèque mathématique est fondé sur le calcul de congruence suivant :

$$u_{n+1} = (\alpha \cdot u_n + \beta) \bmod(m), \text{ avec } \begin{cases} \alpha = 25214903917 \\ \beta = 11 \\ m = 2^{48} \end{cases}$$

Ce générateur fournit des valeurs sur 48 bits (ce qui explique la valeur de m), et il existe des fonctions permettant d'utiliser ces 48 bits pour construire les divers types de données :

```
double drand48 (void);
```

Cette fonction renvoie un réel dans l'intervalle [0, 1[. Comme nous ne disposons que de 48 bits pour remplir un double, dont la fraction normalisée fait 52 bits, les 4 bits de poids faibles sont à 0.

```
double erand48 (unsigned short int etat_generateur [3]);
```

Cette fonction donne le même résultat que `drand48()`, mais elle utilise l'état du générateur représenté par le tableau transmis en argument. Ce dernier est ensuite mis à jour avec le nouvel état.

```
long lrand48 (void);
long nrand48 (unsigned short int etat_generateur [3]);
```

Ces deux fonctions renvoient une valeur entière dans l'intervalle [0, 2^{31}[, même si la taille des `long int` est supérieure à 32 bits. La fonction `nrand48()` utilise l'état transmis et sauvegarde ensuite le nouvel état du générateur.

```
long mrand48 (void);
long jrand48 (unsigned short int etat_generateur [3]);
```

Ces deux fonctions renvoient une valeur entière dans l'intervalle [-2^{31}, 2^{31}[, même si la taille des `long int` est supérieure à 32 bits. La fonction `jrand48()` utilise l'état transmis et sauvegarde ensuite le nouvel état du générateur.

Pour initialiser l'état du générateur aléatoire, plusieurs routines sont disponibles :

```
void srand48 (long int racine);
```

Cette fonction utilise les 32 bits de poids faibles de la racine transmise (même si le type `long` fait plus de 32 bits) pour initialiser les 32 bits de poids forts du générateur. Les 16 bits de poids faibles du générateur prennent la valeur 13 070.

Il s'agit de la routine la plus simple permettant d'initialiser – imparfaitement – le générateur aléatoire.

```
unsigned short int * seed48 (unsigned short int etat [3]);
```

Avec cette routine, on peut définir les 48 bits utilisés comme état du générateur aléatoire. On n'utilise que les 16 bits de poids faibles de chacun des trois `unsigned short` du tableau passé en argument. Le premier élément du tableau sert à initialiser les 16 bits de poids faibles du générateur, le deuxième correspond aux bits 16 à 31, et le troisième contient les 16 bits de poids forts. La fonction renvoie un pointeur sur un tableau contenant l'état précédent. Celui-ci ne sert pas habituellement.

```
void lcong48 (unsigned short int configuration [7]);
```

Cette fonction est la plus complète car elle permet de définir non seulement l'état du générateur, mais également les valeurs α et β de la formule indiquée plus haut.

Les trois premiers éléments du tableau servent à initialiser l'état du générateur aléatoire, comme `seed48()`. Les trois éléments suivants contiennent les 48 bits de la constante α, et le dernier élément comprend les 16 bits de β.

Lorsqu'on appelle `srand48()` ou `seed48()`, les constantes α et β reprennent automatiquement leurs valeurs par défaut.

Les constantes étant stockées dans des variables globales, un problème peut se poser avec des applications multithreads pour lesquelles plusieurs générateurs aléatoires avec des configurations différentes sont nécessaires (encore que le cas soit plutôt rare…). Pour cela, la GlibC offre des extensions Gnu permettant de passer en paramètre les constantes. Ceci se déroule en utilisant un type opaque `struct drand48_data`. Pour pouvoir éventuellement renvoyer une valeur d'erreur (mauvais pointeur par exemple), les fonctions fournissent à présent leur résultat par l'intermédiaire d'un pointeur passé en paramètre.

Le fonctionnement des routines est le même que le précédent, mais les prototypes deviennent :

```
int drand48_r (struct drand48_data * buffer,
               double * resultat);
```

```
int erand48_r (unsigned short int etat [3],
               struct drand48_data * buffer, double * resultat);
int lrand48_r (struct drand48_data * buffer,
               long int * resultat);
int nrand48_r (unsigned short int etat [3],
               struct drand48_data * buffer, long int * resultat);
int mrand48_r (struct drand48_data * buffer,
               long int * resultat);
int jrand48_r (unsigned short int etat [3],
               struct drand48_data * buffer, long int * resultat);
int srand48_r (long int racine,
               struct drand48_data * buffer) ;
int seed48_r  (unsigned short etat [3],
               struct drand48_data * buffer) ;
int lcong48_r (unsigned short configuration [7],
               struct drand48_data * buffer) ;
```

Rappelons que l'utilité de ces routines n'est que ponctuelle. Elles ne servent que si chaque thread a besoin de configurer les constantes α et β de son générateur aléatoire différemment des autres. Si on désire simplement que chaque thread dispose de sa propre séquence, il suffit d'utiliser les fonctions `erand48()`, `nrand48()` ou `jrand48()`. Enfin, si on veut que chaque thread reçoive un nombre aléatoire indépendant des autres, sans qu'une séquence ne se reproduise – ce qui est le cas le plus courant –, on peut utiliser `drand48()`, `lrand48()` ou `mrand48()`.

Conclusion

L'emploi des fonctions mathématiques avec la GlibC peut être motivé par des besoins qui sont nombreux et différents. Pour un complément d'informations, on pourra se reporter par exemple à [KNUTH 1973b] *The Art of Computer Programming* – volume 2, ou à [PRESS 1993] *Numerical Recipes in C*.

On trouvera dans ces ouvrages de nombreuses discussions concernant les nombres aléatoires, la précision des représentations en virgule flottante, les calculs de polynômes, la factorisation, etc.

Pour les programmeurs recherchant des algorithmes géométriques (distance d'un point à une droite, changements de repère, etc.), ce qui représente une utilisation fréquente de la bibliothèque mathématique, signalons que la Faq du groupe Usenet `comp.graphics.algorithms` contient de nombreux renseignements très utiles à cet égard.

25

Fonctions horaires

Il est fréquent qu'une application fasse un usage plus ou moins large de la date ou de l'heure. On peut désirer horodater des enregistrements ou des messages, mémoriser des dates de naissance, vider les données trop vieilles se trouvant en mémoire, ou simplement attendre une dizaine de secondes pour laisser à l'utilisateur la possibilité de réagir et de modifier la configuration par défaut.

Malgré tout, la manipulation des dates est souvent source de problèmes. Il m'a fallu écrire, pour une application de supervision d'un système de radiolocalisation, un module enregistrant diverses statistiques (nombre de trames reçues, états de certains bits d'alarme, etc.). Ces valeurs devaient être mémorisées et cumulées seconde par seconde sur la dernière minute, minute par minute sur la dernière heure, heure par heure sur les dernières vingt-quatre heures, et jour par jour pendant un an. Les complications commencent lorsqu'on sait que les événements à enregistrer n'arrivaient pas nécessairement toutes les secondes mais pouvaient se produire une ou deux fois par mois seulement. Bien entendu, il fallait conserver quand même les statistiques à jour en permanence et pouvoir les afficher à tout moment (en gérant notamment les problèmes dus aux années bissextiles). Ce genre de fonctionnalité devient vite assez acrobatique à élaborer, alors qu'il ne s'agit en réalité que d'une partie accessoire d'un logiciel servant par ailleurs à tout autre chose.

C'est peut-être en cela que la manipulation des données horaires pose des difficultés. Il s'agit souvent de fonctions annexes ou de simples routines d'affichage à l'écran, auxquelles on n'accorde pas toujours l'attention nécessaire. De plus, des cas particuliers peuvent se produire sortant largement du cadre des tests du logiciel. Le problème de l'année bissextile vient bien sûr immédiatement à l'esprit, mais on peut aussi citer l'horloge interne que l'administrateur ramène brutalement en arrière (ce qu'il ne devrait jamais faire, nous le verrons plus bas), ou le processus qui s'est endormi pendant une durée très longue (plusieurs jours) car on a débranché par mégarde un périphérique de communication, etc.

Nous nous intéresserons en premier lieu à la lecture de l'heure et à la configuration de l'horloge interne. Nous examinerons ensuite les fonctions de conversion à utiliser pour afficher des résultats, puis nous aborderons le problème des fuseaux horaires.

Horodatage et type time_t

L'horodatage sous Unix est réalisé à l'aide d'un type de donnée particulier, le type `time_t`. On y stocke le nombre de secondes écoulées depuis le 1er janvier 1970, à 0 heure TU, qu'on considère comme le début de l'ère Unix (*Epoch* en anglais). L'essentiel des datations est accompli à l'aide de ce repère, ce qui rend bien entendu le noyau insensible aux problèmes d'années bissextiles ou de changement de siècle. La norme ISO C9X indique uniquement que le type `time_t` permet des opérations arithmétiques, mais elle ne précise pas qu'il s'agit d'un nombre de secondes. En pratique c'est le cas sur tous les systèmes Unix, mais si on désire vraiment assurer la portabilité d'une manipulation arithmétique horaire, on passera par une conversion intermédiaire en structure `tm` que nous verrons plus loin.

Le type de donnée `time_t` étant exprimé en secondes, il est facile à manipuler car on peut aisément ajouter un délai pour programmer une alarme, sans se soucier du débordement sur la minute, l'heure ou le jour suivant. Traditionnellement, les données `time_t` sont implémentées sous forme d'entiers longs, signés de 32 bits. C'est le cas pour l'essentiel des implémentations de Linux, hormis les architectures SPARC. Ceci permet donc de gérer des dates jusqu'à un maximum de 0x7FFFFFFF, soit 2 147 483 647 secondes depuis le 1er janvier 1970. Malheureusement, ce nombre n'est pas aussi énorme qu'il en a l'air. Le mardi 19 janvier 2038 à 3 heures 14 minutes et 7 secondes TU, les compteurs `time_t` 32 bits signés, s'il en reste, basculeront à 0x80000000, soit –2 147 483 648 secondes, et reviendront donc au vendredi 13 décembre 1901, à 20 heures 45 minutes et 52 secondes T.U. !

Bien sûr, cela n'arrivera pas réellement, car d'ici là les noyaux Unix seront mis à jour pour traiter les données `time_t` avec un autre stockage, probablement un 64 bits signés. Le problème qui se pose toutefois est l'interface des applications fonctionnant sur ces systèmes. Car si le noyau modifie la longueur du type `time_t`, cela vaudra également pour la bibliothèque C et les applications qui utilisent les fonctions de lecture d'heure que nous verrons ci-dessous.

Dans l'immense majorité des cas, une simple recompilation permettra de mettre à niveau le logiciel. Toutefois, le cas des applications disponibles uniquement sous forme binaire posera un problème essentiel, ainsi que pour les systèmes gérant des bases de données dans lesquelles les dates sont stockées, avec `fwrite()` par exemple, de manière binaire dans des fichiers. Il sera nécessaire d'écrire des outils de conversion des bases de données.

L'an 2038 peut paraître bien éloigné aujourd'hui. Une bonne partie des informaticiens actuels ne seront plus en activité à ce moment-là, aussi le problème ne leur semble pas aussi crucial que cela. Pourtant, ce raisonnement est faux pour plusieurs raisons :

- Le vent de panique créé lors du passage à l'an 2000 devrait nous servir de leçon pour savoir qu'on ne peut pas prédire la durée de vie d'une application. Elle peut non seulement être utilisée bien plus longtemps que ce qu'on estimait lors de son écriture, mais cela semble encore plus vrai pour les logiciels dont les sources ne sont pas disponibles.
- Il existera de plus en plus de systèmes embarqués, qu'on trouvera dans les appareils électroménagers, hi-fi, voitures, appareils photographiques numériques… Le logiciel embarqué sera de plus en plus évolué, et une bonne partie sera constituée d'un véritable noyau Unix sur lequel tournera l'application faisant fonctionner le matériel, mais également des outils

de communication, pour la programmation, le paramétrage ou l'évolution du logiciel. La durée de vie de ces appareils pourra être très longue, et des applications conçues dans un avenir proche pourront fort bien continuer d'exister dans des équipements fonctionnant toujours en 2038.

- De nombreux programmes n'ont pas besoin d'attendre 2038 pour être confrontés à ce problème. Un logiciel de calcul astronomique peut par exemple être employé pour prévoir des événements 10, 15, 20 ans à l'avance. Il en est de même pour un programme faisant des calculs d'amortissements pour un emprunt sur 20 ou 25 ans. Il devra alors faire face au bogue de 2038 dès l'an 2013. Le délai restant est alors largement diminué.

Pour toutes ces raisons, il est important pour un programmeur applicatif de commencer à se préoccuper de l'utilisation qu'il fait des données `time_t`. Les manipulations internes dans le programme ne posent en fait pas vraiment de problème. Une recompilation du logiciel permettra de prendre en compte la nouvelle longueur lorsqu'il le faudra.

Les difficultés s'annoncent lorsqu'on doit stocker des dates dans un fichier ou les communiquer à un autre système. Dans un cas comme dans l'autre, si on est maître des deux extrémités de la transmission (lecture et écriture du fichier, ou émission et réception des données), on peut employer un subterfuge consistant à transférer les données de type `time_t` dans un entier `long long int`, qui dispose au moins de 64 bits sur les systèmes actuels. Ce sera alors cette variable qui sera utilisée pour la transmission ou le stockage. La conversion inverse supprimera les bits supplémentaires, inutilisés à ce moment-là, tant que le type `time_t` n'aura pas évolué.

Si cette solution n'est pas possible, il faut se contenter de bien documenter par des commentaires précis les emplacements où la taille des données `time_t` est prise en compte. Les évolutions qui permettront de basculer sur un type plus long ne sont pas encore prévisibles. Peut-être verra-t-on apparaître un type `time64_t` intermédiaire ou une utilisation des 32 bits de manière non signée, ce qui permettrait de gagner près de 70 ans de plus[1].

Lecture de l'heure

L'appel-système le plus simple pour lire l'heure actuelle est **`time()`**, que nous avons déjà observé rapidement dans le chapitre 9, et qui est déclaré dans `<time.h>` :

```
time_t time(time_t * heure);
```

Cet appel-système renvoie la date et l'heure actuelles, et remplit la variable transmise en argument avec cette même valeur si le pointeur n'est pas `NULL`. Si jamais le pointeur est invalide, `time()` retourne la valeur d'erreur `((time_t) -1)`. Cet appel-système est simple, portable – défini par SUSv3 et Ansi C –, et nous avons vu que le type de donnée `time_t` est facile à manipuler.

Il peut arriver cependant qu'on ait besoin de dater des événements avec une précision plus grande que la seconde. Pour cela, il existe un appel-système fournissant une meilleure résolution. L'appel **`gettimeofday()`** est déclaré dans `<sys/time.h>`, ainsi que les types des données qu'il emploie :

```
int gettimeofday (struct timeval * timev, struct timezone * timez);
```

1. Le type `time_t` n'est pas nécessairement signé, il faut simplement que `((time_t) -1)` ait une signification. Cela est possible même avec un entier non signé, par exemple `((unsigned char) -1)` vaut `255`. La valeur `0xFFFFFFFF` sera donc une valeur d'erreur.

Cette fonction remplit les deux structures sur lesquelles on passe des pointeurs – s'ils ne sont pas `NULL` –, et renvoie 0 si elle réussit et –1 en cas d'erreur. La structure `timeval`, déjà vue à plusieurs reprises avec `wait3()`, `setitimer()` ou encore `select()`, contient les deux membres suivants :

Nom	Type	Signification
`tv_sec`	`time_t`	Nombre de secondes écoulées depuis le 1er janvier 1970
`tv_usec`	`time_t`	Nombre de microsecondes depuis le dernier changement de `tv_sec`

Bien entendu, on pourrait construire la fonction `time()` à partir de `gettimeofday()` ainsi :

```
    time_t
time (time_t * timer)
{
    struct timeval timev;
    gettimeofday (& timev, NULL);
    if (timer != NULL)
        * timer = timev.tv_sec;
    return timev.tv_sec;
}
```

Toutefois, sous Linux, l'implémentation est encore sous forme d'appel-système indépendant, ce qui présente par ailleurs l'avantage d'une meilleure vérification de la validité du pointeur transmis.

La structure `timezone` contient deux membres :

Nom	Type	Signification
`tz_minuteswest`	`int`	Nombre de secondes de décalage vers l'ouest depuis Greenwich
`tz_dsttime`	`int`	Type d'horaire hiver / été appliqué localement

La structure `timezone` est quasi obsolète et ne doit pas être utilisée. Nous verrons à la fin de ce chapitre comment accéder aux informations sur les fuseaux horaires. Le premier membre de `timezone` peut indiquer correctement la bonne valeur, mais le second n'est jamais mis à jour. Dans la plupart des cas, on n'utilisera jamais le second argument de `gettimeofday()`, et on passera donc un pointeur `NULL`.

Voyons donc les comportements de `time()` et de `gettimeofday()` :

exemple_gettimeofday.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    #include <unistd.h>
    #include <sys/time.h>

    int
main (void)
{
    struct timeval timev;
```

```
    if (gettimeofday(& timev, NULL) != 0) {
        perror("gettimeofday");
        exit(EXIT_FAILURE);
    }
    fprintf (stdout, "time() : %ld \n", time (NULL));
    fprintf (stdout, "gettimeofday() : %ld.%06ld\n",
            timev . tv_sec, timev . tv_usec);
    return (0);
}
```

L'exécution montre bien le même résultat au niveau seconde et une meilleure précision de `gettimeofday()`.

```
$ ./exemple_gettimeofday
time() : 1105008472
gettimeofday() : 1105008472.756992
$
```

Nous mentionnerons également l'existence d'une fonction obsolète nommée **`ftime()`**, déclarée dans `<sys/timeb.h>` :

```
int ftime (struct timeb * timeb);
```

La structure `timeb` regroupait en fait les champs des structures `timeval` et `timezone` ainsi :

Nom	Type	Équivalence
time	time_t	timeval . tv_sec
millitm	unsigned short int	timeval . tv_usec
timezone	short int	timezone . tz_minuteswest
dstflag	short int	timezone . tz_dsttime

Configuration de l'heure système

Le réglage de l'heure du système est une opération évidemment privilégiée, nécessitant un UID effectif nul ou la capacité `CAP_SYS_TIME`. Il existe trois appels-système permettant de modifier l'heure de la machine : **`settimeofday()`**, qui est un héritage de BSD, **`stime()`**, qui provient de Système V, et **`adjtimex()`**, qui est spécifique à Linux. Leurs prototypes sont déclarés respectivement dans `<sys/time.h>`, `<time.h>` et `<sys/timex.h>` ainsi :

```
int settimeofday (const struct timeval * timeval,
                  const struct timezone * timezone);
int stime (time_t * heure);
int adjtimex (struct timex * timex);
```

L'appel-système `settimeofday()` fonctionne à l'inverse de `gettimeofday()`, en configurant l'heure et éventuellement le fuseau horaire de la machine.

L'appel `stime()` peut très bien être implémenté à partir de `settimeofday()`, comme nous l'avons observé pour son antagoniste `time()`.

Enfin, `adjtimex()` sert non seulement à régler l'heure de l'horloge interne, mais permet aussi d'organiser des paramètres complexes pour ajuster la régularité de l'horloge et éviter des dérives périodiques. Ce sujet sort largement du cadre de notre étude, et nous laisserons le lecteur que

cela intéresse se reporter directement aux sources du noyau, en étudiant les fichiers `kernel/time.c` et `arch/xxx/kernel/time.c`, ou à la RFC 956, qui décrit le principe de cet algorithme.

Il est fortement déconseillé d'utiliser directement ces appels-système. En effet, l'horloge du noyau doit fonctionner de la manière la plus monotone possible. Modifier brutalement l'heure du système ou, pire encore, la faire revenir en arrière peuvent perturber gravement certains processus qui traitent des données horodatées. Pour configurer l'horloge de la machine, on préférera employer la fonction de bibliothèque **`adjtime()`**, spécifique à l'extension Gnu et déclarée dans :

```
int adjtime (const struct timeval * modification,
             struct timeval * ancienne);
```

Cette fonction prend en premier argument un pointeur sur une structure `timeval` contenant la *différence* entre l'heure désirée et l'heure actuelle. Cette différence peut notamment être négative, si on désire retarder l'horloge. La bibliothèque C va alors ralentir l'horloge système de manière à rattraper progressivement la valeur voulue. De même, lorsque la différence est positive, l'horloge sera accélérée pour combler peu à peu l'écart. Si le second argument est un pointeur non `NULL`, on y stocke la modification précédemment demandée et qui n'a pas fini d'être appliquée.

Cette fonction est très précieuse par exemple pour synchroniser plusieurs machines d'un réseau local en employant le protocole NTP (défini dans la RFC 1305).

Conversions, affichages de dates et d'heures

Pour le moment nous n'avons manipulé la date et l'heure que sous forme de données de type `time_t` (ou de structures `timeval` qui l'encadrent en ajoutant les microsecondes). Nous avons observé que ce type est pratique (l'unité étant la seconde, il est très intuitif), robuste (pas de problème d'années bissextiles ou de changement de siècle), et portable (défini par SUSv3 et Ansi C). Toutefois, malgré tous ces avantages, on arrive difficilement à faire comprendre à l'utilisateur que 947846794 est plus commode que 14 janvier 2000 à 11 heures 46 minutes et 34 secondes. Il faut donc trouver le moyen de convertir les secondes du type `time_t` en éléments plus lisibles par un utilisateur moyen. La bibliothèque C nous fournit plusieurs routines de traduction.

Tout d'abord, il existe une structure de données permettant de représenter la date et l'heure sous forme intelligible. La structure **`tm`** est définie par le standard Ansi C et contient les membres suivants, qui sont tous de type `int` :

Nom	Signification
`tm_sec`	Nombre de secondes écoulées depuis le dernier changement de minute, dans l'intervalle 0 à 60
`tm_min`	Nombre de minutes écoulées depuis le dernier changement d'heure, entre 0 et 59
`tm_hour`	Nombre d'heures écoulées depuis minuit, dans l'intervalle 0 à 23
`tm_mday`	Jour du mois, allant de 1 à 31
`tm_mon`	Nombre de mois écoulés depuis le début de l'année, dans l'intervalle 0 à 11
`tm_year`	Nombre d'années écoulées depuis 1900
`tm_wday`	Nombre de jours écoulés depuis dimanche dans l'intervalle 0 à 6
`tm_yday`	Nombre de jours écoulés depuis le 1er janvier, dans l'intervalle 0 à 365
`tm_isdst`	Indicateur d'horaire d'été ou d'hiver

Plusieurs points appellent des commentaires dans ce tableau :

- Les secondes peuvent aller de 0 à 60, car il existe parfois des secondes de rattrapage périodique, définies par les instances astronomiques internationales. Une minute officielle peut donc durer 59, 60 ou 61 secondes. En réalité, les fonctions de la bibliothèque C ne renvoient jamais de valeur supérieure à 59, comme cela est demandé par SUSv3 (elles arrondissent au besoin à la minute supérieure). Par contre, on peut légitimement remplir le champ `tm_sec` avec une valeur allant jusqu'à 60 en entrée des fonctions de la GlibC. Un programme portable devra donc être prêt à traiter ce cas, par exemple s'il utilise les secondes comme index dans un tableau de statistiques. Il faudra alors soit prévoir 61 emplacements, soit utiliser une astuce comme `index=(tm.tm_sec % 60)` ou `index =(tm.tm_sec < 60 ? tm.tm_sec : 59)`.
- Le jour du mois commence à 1 et n'est donc pas directement utilisable comme index dans une table, mais il peut être affiché. Par contre, le numéro du mois débute à zéro. Il faut lui ajouter 1 pour l'affichage.
- Le membre `tm_year` indique le nombre d'années écoulées depuis 1900. L'an 2000 est donc représenté par un 100. Pour afficher l'année sur deux chiffres, on emploiera donc `(tm_year % 100)`. Ceci ne pose plus de problème pour les nouvelles applications puisque en cas d'erreur le problème apparaîtra dès les premiers tests avec par exemple un affichage 25/12/101. Par contre, de nombreux logiciels conçus jusqu'en 1999 peuvent souffrir d'un défaut d'attention du programmeur face à cette caractéristique.
- La semaine commence, à l'anglaise, le dimanche et pas le lundi. Le champ `tm_wday` va de 0 à 6, pouvant servir d'index dans un tableau initialisé ainsi : `char * jours[7]={"D","L", "Ma", "Me", "J", "V", "S"};`
- Le membre `tm_isdst` a une valeur positive si l'horaire d'été est en vigueur, nulle si l'horaire normal (hiver) fonctionne, et négative si cette information n'est pas disponible.

La bibliothèque GlibC ajoute également deux autres membres `tm_gmtoff` et `tm_zone`, qui correspondent respectivement au nombre de secondes qu'il faut ajouter à la date indiquée pour obtenir le temps TU, et au nom (sous forme de chaîne de caractères statique) du fuseau horaire employé. Ces deux champs ne sont pas standard et nous ne les traiterons pas ici.

Les routines de conversion de format de date renvoient traditionnellement des pointeurs sur des zones de mémoire allouées statiquement. Ces données sont donc écrasées à chaque nouvel appel de la même fonction. Ceci rend impossible leur utilisation dans un contexte multithread. Aussi la bibliothèque GlibC inclut-elle des extensions Unix 98 avec le suffixe `_r` pour définir une version réentrante de chacune de ces routines.

Pour convertir une valeur de type `time_t` en structure `tm`, il existe deux fonctions, **`localtime()`** et **`gmtime()`**, et leurs homologues réentrantes :

```
struct tm * localtime   (const time_t * date);
struct tm * localtime_r (const time_t * date, struct tm * tm);
struct tm * gmtime      (const time_t * date);
struct tm * gmtime_r    (const time_t * date, struct tm * tm);
```

Bien entendu, les deux premières routines renvoient l'heure locale, en se fondant sur la configuration des fuseaux horaires que nous verrons plus bas, alors que les deux dernières retournent l'heure TU.

Voici un exemple d'emploi de ces routines :

exemple_localtime.c :

```
    #include <stdio.h>
    #include <time.h>

    int
main (void)
{
    time_t      temps;
    struct tm * tm;

    time(& temps);
    fprintf(stdout, "time() = %ld \n", temps);
    tm = localtime(& temps);
    fprintf(stdout, "localtime() = %02d/%02d/%02d - %02d:%02d:%02d %s\n",
        tm->tm_mday, tm->tm_mon + 1, tm->tm_year % 100,
        tm->tm_hour, tm->tm_min, tm->tm_sec,
        tm->tm_isdst>0 ? "Été" : tm->tm_isdst==0 ? "Normal" : "?");
    tm = gmtime (& temps);
    fprintf(stdout, "gmtime()    = %02d/%02d/%02d - %02d:%02d:%02d %s\n",
        tm->tm_mday, tm->tm_mon + 1, tm->tm_year % 100,
        tm->tm_hour, tm->tm_min, tm->tm_sec,
        tm->tm_isdst>0 ? "Été" : tm->tm_isdst==0 ? "Normal" : "?");
    return EXIT_SUCCESS;
}
```

Les exécutions suivantes du programme ont lieu dans le fuseau horaire de Paris :

```
$ ./exemple_localtime
time() = 932303050
localtime() = 18/07/99 - 15:04:10 Été
gmtime()    = 18/07/99 - 13:04:10 Normal
$
```

L'horaire d'été est bien détecté, voyons l'horaire d'hiver :

```
$ ./exemple_localtime
time() = 947855103
localtime() = 14/01/00 - 14:05:03 Normal
gmtime()    = 14/01/00 - 13:05:03 Normal
$
```

La traduction inverse est possible, grâce à la fonction **`mktime()`** :

```
time_t mktime(struct tm * tm);
```

Cette routine peut renvoyer `(time_t)-1` en cas d'erreur, mais elle essaye toutefois d'être la plus robuste possible. Elle ignore les membres `tm_yday` et `tm_wday` de la structure `tm` transmise, elle les recalcule grâce aux autres données et les remet à jour. Si un membre a une valeur invalide, la fonction `mktime()` calcule son débordement. Par exemple, 23h70 est corrigé pour correspondre à 0h10 du jour suivant.

On peut bien entendu utiliser une fonction de la famille `printf()` pour présenter le contenu d'une structure `tm`, comme nous l'avons fait ci-dessus, mais lorsqu'on désire afficher la date

uniquement à titre informatif pour l'utilisateur, il est souvent plus simple d'utiliser l'une des fonctions **asctime()** et **ctime()**, qui renvoient des chaînes de caractères statiques, ou leurs homologues **asctime_r()** et **ctime_r()**, qui utilisent un buffer passé en argument, pouvant contenir au minimum 16 caractères.

```
char * asctime   (const struct tm * tm);
char * asctime_r (const struct tm * tm, char * buffer);
char * ctime     (const time_t * date);
char * ctime_r  (const time_t * date, char * buffer);
```

La fonction `ctime()` est l'équivalent de `asctime (localtime (date))`. Le résultat de ces fonctions est une chaîne de caractères contenant :

- Le jour de la semaine, parmi les abréviations Mon, Tue, Wed, Thu, Fri, Sat, Sun ;
- Le nom du mois parmi Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec ,
- Le numéro du jour dans le mois ;
- L'heure, les minutes et les secondes ;
- L'année sur quatre chiffres ;
- Un caractère '`\n`' de retour à la ligne.

En voici une illustration très simple.

exemple_ctime.c :

```
    #include <stdio.h>
    #include <time.h>

    int
main (void)
{
    time_t t;
    t = time(NULL);
    fprintf(stdout, "%s", ctime(& t));
    return EXIT_SUCCESS;
}
```

```
$ ./exemple_ctime
Thu Jan  6 11:54:44 2005
$
```

Nous voyons qu'avec `ctime()` ou `asctime()` le format d'affichage est figé. De plus, le nom des jours et des mois est en anglais. Ces routines ne sont pas sensibles à la localisation du processus. Pour pallier ces problèmes, la bibliothèque C propose une routine définie par SUSv3, **strftime()**, très puissante mais légèrement plus compliquée puisqu'elle fonctionne un peu sur le principe de la famille `printf()`.

```
size_t strftime (char * buffer, size_t longueur,
                 const char * format, const struct tm * tm);
```

Cette fonction remplit le buffer passé en premier argument, dont la taille est indiquée en second argument. Si ce buffer est trop court, `strftime()` renvoie 0. Sinon, elle transmet le nombre de caractères écrits, sans compter le '`\0`' final. Le contenu du buffer est constitué des

champs de la structure `tm` indiqués dans la chaîne de format passée en troisième argument. Les codes employés sont indiqués dans le tableau suivant :

Code	Signification
`%%`	Le caractère `%`.
`%A`	Le nom complet du jour de la semaine.
`%a`	Le nom abrégé du jour de la semaine.
`%B`	Le nom complet du mois.
`%b`	Le nom abrégé du mois.
`%C`	Le siècle (19 pour 1977, 20 pour 2015).
`%c`	La date et l'heure dans la représentation locale usuelle.
`%D`	La date, dans le format `%m/%d/%y`.
`%d`	Le jour du mois dans l'intervalle 1 à 31.
`%e`	Le jour du mois dans l'intervalle 1 à 31, précédé par un blanc pour les valeurs inférieures à 10, afin de permettre un alignement à droite.
`%F`	La date, dans le format `%Y-%m-%d`. Ce format est défini par la norme ISO 8601, il deviendra probablement de plus en plus répandu dans l'avenir.
`%g`	Le numéro de l'année sur deux chiffres, correspondant à la semaine en cours. Il peut y avoir une différence avec `%y` pour les premiers ou derniers jours de l'année.
`%G`	Comme `%g`, mais sur quatre chiffres.
`%h`	Comme `%b`.
`%H`	L'heure sur 24 heures et sur deux chiffres, de 00 à 23.
`%I`	L'heure sur 12 heures et sur deux chiffres, de 00 à 11.
`%j`	Le numéro du jour de l'année sur trois chiffres, de 001 à 366.
`%k`	L'heure sur 24 heures, mais avec un espace devant les valeurs inférieures à 10, allant donc de 0 à 23. Extension Gnu.
`%l`	Comme `%k`, mais sur 12 heures. Extension Gnu.
`%M`	La minute sur deux chiffres, de 00 à 59.
`%m`	Le numéro du mois, sur deux chiffres, de 01 à 12.
`%n`	Un caractère '`\n`' de retour à la ligne.
`%P`	Comme `%p`, mais en majuscules. Extension Gnu.
`%p`	L'équivalent local des chaînes «`am`» ou «`pm`» de l'heure sur 12 heures. Minuit est considéré comme 0h am et midi comme 0h pm.
`%R`	L'heure et la minute au format `%H:%M`. Extension Gnu.
`%r`	L'heure complète, sur 12 heures, y compris les équivalents locaux de am et pm.
`%S`	Les secondes sur deux chiffres, de 00 à 60.
`%s`	Le nombre de secondes écoulées depuis le 1er janvier 1970 à 0 heure TU. Extension Gnu.
`%T`	L'heure au format `%H:%M:%S`.
`%t`	Le caractère '`\t`' de tabulation.

Code	Signification
%U	Le numéro de la semaine dans l'année, allant de 00 à 53. La semaine numéro 1 de l'année commence au premier dimanche. Les jours précédant ce dimanche sont dans la semaine 0.
%u	Le numéro du jour dans la semaine, de 1 à 7, avec 1 correspondant au lundi.
%V	Le numéro de la semaine dans l'année de 1 à 53. La référence prise ici commence au premier lundi de l'année, comme le précise la norme ISO 8601.
%W	Comme %V, mais de 0 à 53, les jours précédant le premier lundi étant dans la semaine 0.
%w	Le jour de la semaine de 0 à 6, en commençant le dimanche.
%X	La représentation locale usuelle de l'heure.
%x	La représentation locale usuelle de la date. Le compilateur nous avertit lorsqu'on utilise %x dans une chaîne constante que cette représentation se fait avec des années sur deux chiffres dans certaines localisations. Cet avertissement peut être ignoré si on est conscient de ce fait.
%Y	L'année sous forme de nombre décimal complet.
%y	L'année sur deux chiffres, sans le siècle.
%Z	Le nom du fuseau horaire abrégé, éventuellement vide.
%z	Le fuseau horaire indiqué sous forme numérique conforme à la RFC 822.

Lorsqu'on transmet un buffer NULL, strftime() nous indique le nombre de caractères qu'elle aurait dû écrire dedans. Ceci est très utile car, dans certaines situations, cette fonction peut renvoyer légitimement 0 alors que le buffer est bien assez grand. C'est le cas par exemple si on demande uniquement le code %p alors que la localisation permet seulement l'emploi du temps sur 24 heures. Nous allons employer cette méthode dans le programme suivant.

exemple_strftime.c :

```
    #include <limits.h>
    #include <locale.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>

    int
main (int argc, char * argv[])
{
    int         i;
    int         lg;
    char *      buffer;
    struct tm * tm;
    time_t      heure;

    setlocale(LC_ALL, "");

    time(& heure);
    tm = localtime(& heure);

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "%s : ", argv[i]);
```

```
        lg = strftime(NULL, SSIZE_MAX, argv[1], tm);
        if (lg > 0) {
            buffer = malloc(lg + 1);
            if (buffer == NULL) {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
            strftime(buffer, lg + 1, argv[i], tm);
            fprintf(stdout, "%s", buffer);
            free(buffer);
        }
        fprintf(stdout, "\n");
    }
    return EXIT_SUCCESS;
}
```

Ce programme permet d'afficher la date et l'heure courantes avec le format transmis en argument de la ligne de commande. En voici quelques exemples :

```
$ ./exemple_strftime "Le %d %B %Y, à %H heures %M"
Le %d %B %Y, à %H heures %M : Le 06 janvier 2005, à 11 heures 59
$ ./exemple_strftime %p
%p :
$ ./exemple_strftime "%Z (%z)"
%Z (%z) : CET (+0100)
$
```

Nous remarquons que dans la localisation française, le code `%p` (AM / PM) n'a pas de signification.

Bien entendu, la bibliothèque GlibC propose des fonctions permettant le cheminement inverse, c'est-à-dire la création d'une structure `tm` à partir d'une chaîne de caractères qui peut avoir été saisie par l'utilisateur. Deux fonctions existent, **`strptime()`** et `getdate()`, toutes deux déclarées dans `<time.h>` et définies par SUSV3.

```
char * strptime (const char * chaine_lue, const char * format,
                 struct tm * tm);
```

Cette routine fonctionne un peu comme `sscanf()`. Elle examine le contenu de la chaîne transmise en premier argument, à la lumière du format précisé en second argument. Le résultat est alors stocké dans la structure `tm`, puis renvoie un pointeur sur le premier caractère de la chaîne initiale qui n'a pas été converti.

La mise en correspondance entre la chaîne lue et le format est faite octet par octet, chaque caractère du format devant avoir un équivalent dans la chaîne, sinon la lecture s'arrête. Bien entendu, des codes spéciaux identiques à ceux de `strftime()` peuvent être insérés pour lire les champs de la structure `tm`. Les codes étant les mêmes, la fonction `strptime()` est donc symétrique à `strftime()`, et peut aussi bien être employée pour relire des données écrites par un programme que pour lire une saisie humaine.

Si toute la chaîne a pu être analysée, le pointeur transmis correspond au caractère nul final, '`\0`'. Par contre, si aucune conversion n'a pu avoir lieu, le pointeur est `NULL`. Il faut donc systématiquement vérifier cette condition avant d'essayer de consulter le contenu du pointeur, sous peine de déclencher une erreur `SIGSEGV`.

La bibliothèque GlibC met à jour uniquement les champs de la structure `tm` qui ont été lus, ainsi que les champs `tm_wday` et `tm_yday`. Les autres membres ne sont pas modifiés. Pour vérifier le résultat de la fonction, il est donc conseillé d'initialiser tous les membres avec une valeur impossible, comme –1 ou `INT_MAX`. Cela permettra de s'assurer de la réussite de la conversion. Si on veut éviter cette étape, on peut éventuellement initialiser tous les membres avec des zéros, ainsi la structure aura toujours un contenu cohérent.

Pour que `strptime()` soit déclarée dans `<time.h>`, il faut définir la constante symbolique `_XOPEN_SOURCE` avant l'inclusion de cet en-tête.

Le programme suivant va lire la ou les chaînes de formats successifs sur sa ligne de commande, en afficher un exemple sur `stdout`, et demander à l'utilisateur une saisie sur `stdin`. La même structure `tm` sera utilisée tout au long des saisies. Ensuite, le résultat sera affiché au complet.

exemple_strptime.c :

```
    #define  _XOPEN_SOURCE
    #include <limits.h>
    #include <locale.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>

    int
main (int argc, char * argv[])
{
    int         i;
    int         lg;
    time_t      heure;
    struct tm   tm;
    struct tm * exemple;
    char *      buffer;
    char *      retour;

    setlocale(LC_ALL, "");

    time(& heure);
    exemple = localtime(& heure);
    memset(& tm, 0, sizeof(struct tm));

    for (i = 1; i < argc ; i ++) {
        lg = strftime(NULL, SSIZE_MAX, argv[i], exemple);
        if (lg > 0) {
            /* On alloue 2 octets de plus pour \n et \0 */
            buffer = malloc(lg + 2); /* retour à vérifier...*/
            strftime(buffer, lg + 2, argv[i], exemple);
            fprintf(stdout, "Format %s (exemple %s) : ",
                            argv[i], buffer);
            while (1) {
                fgets(buffer, lg + 2, stdin);
                retour= strptime(buffer, argv[i], & tm);
                if (retour == NULL)
                    fprintf(stdout, "Erreur > ");
```

```
                else
                    break;
            }
            free(buffer);
        }
    }
    puts(asctime (& tm));
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution simple, mais intéressant à plusieurs égards :

```
$ ./exemple_strptime "Le %x" "à %X"
Format Le %x (exemple Le 16.01.2000) : Le 4.7.1967
Format à %X (exemple à 20:09:52) : à 4:20:0
Tue Jul  4 04:20:00 1967

$ ./exemple_strptime %F %r
Format %F (exemple 2000-01-16) : 2222-2-2
Format %r (exemple 08:11:11 ) : 20:12:10
Erreur > 20:10
Erreur > 08:10:10
Erreur > 08:10:10 PM
Erreur >  (Contrôle-C)
$
```

Nous remarquons plusieurs choses :

- Les versions précédentes de la bibliothèque C ne savaient pas calculer les jours pour les dates antérieures au 1er janvier 1970. Ce défaut est de nos jours corrigé.
- Le format `%r` pose des problèmes car, dans le cas d'une localisation française, la chaîne AM/PM est indéfinie. Lors d'un affichage avec `strftime()`, tout est masqué car une chaîne vide est affichée, mais lors d'une écriture, la mise en correspondance n'est pas possible. Ceci peut engendrer de sérieux problèmes, qui n'apparaîtront que lors de l'exportation d'une application.
- On peut également regretter l'absence de code d'erreur indiquant le type de problème qui s'est présenté.

La bibliothèque C met donc à notre disposition la fonction **`getdate()`** et sa correspondante **`getdate_r()`**, qui peuvent simplifier la lecture des chaînes contenant des données d'horodatage :

```
struct tm * getdate (const char * chaine_lue);
int getdate_err;
int getdate_r (const char * chaine_lue, struct tm * tm);
```

La fonction `getdate()` analyse la chaîne transmise et renvoie un pointeur vers une structure `tm` statique représentant la date obtenue. En cas d'erreur, elle retourne un pointeur `NULL` et positionne la variable globale `getdate_err` avec un code d'erreur détaillé ci-dessous. La fonction `getdate_r()` n'emploie pas de structure statique, mais utilise le pointeur passé en second argument. En conséquence, elle renvoie un code de retour signalant les conditions d'erreur, mais n'utilise pas la variable `getdate_err`.

Pour réaliser l'analyse de la chaîne, ces routines utilisent la variable d'environnement DATEMSK. Celle-ci doit contenir le chemin d'accès et le nom d'un fichier comprenant des motifs de conversion identiques à ceux qui sont employés par strptime(). Chaque motif possible est présenté sur une ligne du fichier, et ils sont essayés successivement jusqu'à ce que l'un d'eux soit correct. L'utilisateur a donc la possibilité de configurer le format de la conversion en fonction de ses habitudes (ou du logiciel employé pour fournir les données d'entrée alimentant la routine getdate() concernée).

En contrepartie, la possibilité d'indiquer soi-même le fichier contenant les motifs à utiliser peut constituer une faille de sécurité dans un programme Set-UID (car on peut alors consulter n'importe quel fichier du système, y compris /etc/shadow dont nous parlerons dans le prochain chapitre). Dans un tel cas, on évitera d'employer getdate() ou on figera lors de la compilation le contenu de la variable d'environnement DATEMSK, en utilisant la routine setenv() étudiée au chapitre 3.

Les codes d'erreur transmis par getdate() dans getdate_err ou renvoyés par getdate_r() sont :

Valeur	Signification
0	Pas d'erreur.
1	Variable DATEMSK non configurée ou contenant une chaîne vide.
2	Le fichier de motifs indiqué dans DATEMSK ne peut pas être ouvert.
3	L'état du fichier de motifs n'est pas accessible.
4	Le fichier de motifs n'est pas un fichier régulier.
5	Impossible de lire le contenu du fichier de motifs.
6	Pas assez de mémoire disponible.
7	Impossible de trouver un motif permettant de réaliser une conversion correcte.
8	La chaîne contient des données invalides après conversion (par exemple 31 avril).

Pour que les prototypes de ces routines soient présents dans <time.h>, il faut remplir la constante symbolique _XOPEN_SOURCE avec la constante 500 avant d'inclure ce fichier d'entête.

Les membres de la structure tm qui ne sont pas renseignés par la chaîne fournie sont initialisés avec la date et l'heure de l'appel de la routine. Voici un programme qui emploie getdate() pour analyser les chaînes transmises en ligne de commande.

exemple_getdate.c :

```
    #define _XOPEN_SOURCE 500
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>

    int
main (int argc, char * argv[])
{
struct tm *  tm;
int          i;
```

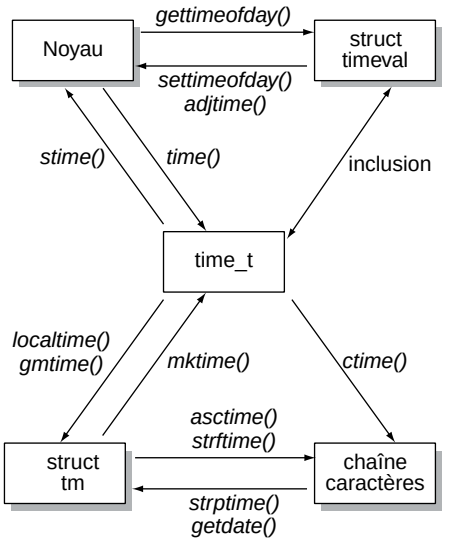
```
    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "%s : ", argv[i]);
        tm = getdate(argv[i]);
        if (tm == NULL)
            switch (getdate_err) {
                case 1 :
                    fprintf(stdout, "DATEMSK indéfinie \n");
                    break;
                case 2 :
                case 3 :
                case 4 :
                case 5 :
                    fprintf(stdout, "Fichier de motifs invalide \n");
                    break;
                case 6 :
                    fprintf(stdout, "Pas assez de mémoire \n");
                    break;
                case 7 :
                    fprintf(stdout, "Conversion impossible \n");
                    break;
                case 8 :
                    fprintf(stdout, "Valeur invalide \n");
                    break;
            }
        else
            fprintf(stdout, "%s", asctime(tm));
    }
    return EXIT_SUCCESS;
}
```

Nous créons un fichier de motifs nommé `datemsk.txt`, qui contient plusieurs conversions possibles sur le thème de la date et de l'heure. Voici un exemple de quelques exécutions :

```
$ cat datemsk.txt
%F %H:%M:%S
%F %H:%M
%F
$ ./exemple_getdate 2000-01-14
2000-01-14 : DATEMSK indéfinie
$ export DATEMSK=datemsk.txt
$ ./exemple_getdate 2000-01-14
2000-01-14 : Fri Jan 14 18:31:25 2000
$ ./exemple_getdate "2000-01-14 05"
2000-01-14 05 : Conversion impossible
$ ./exemple_getdate "2000-01-14 05:06"
2000-01-14 05:06 : Fri Jan 14 05:06:00 2000
$ ./exemple_getdate "2000-01-14 05:06:07"
2000-01-14 05:06:07 : Fri Jan 14 05:06:07 2000
$ ./exemple_getdate "2000-04-31"
2000-04-31 : Valeur invalide
$
```

La possibilité de modifier soi-même le format des conversions est une fonctionnalité très puissante. Indiquons toutefois que `getdate()`, ou plutôt la fonction `strptime()` sous-jacente, n'est pas très robuste vis-à-vis de la localisation, et que l'application risque parfois de planter sur une faute de segmentation, notamment en employant les codes `%r`, `%X` ou `%x`.

Figure 25.1
Conversions de données d'horodatage



La figure 25.1 récapitule toutes les routines de conversion que nous avons vues afin de passer d'une représentation d'une date à une autre.

Calcul d'intervalles

Les fonctions que nous avons étudiées permettent de travailler sur un instant précis. Pourtant, il est parfois indispensable de travailler sur des durées, sur des intervalles. Nous pouvons par exemple avoir besoin d'ajouter à l'heure actuelle un délai maximal de réaction afin de programmer une alarme.

Rappelons que les routines `timeradd()`, `timerclear()`, `timersub()` et `timerisset()`, déjà présentées dans le chapitre 9, permettent une manipulation facile des structures `timeval`, en garantissant que le membre `tv_usec` sera toujours compris entre 0 et 999 999, ce qui est obligatoire mais pas toujours facile à conserver.

Le type `time_t` étant signé et contenant des secondes, il est possible d'ajouter ou de soustraire des durées sans problème. Toutefois, si ce type de donnée devait évoluer, comme nous l'avons évoqué au début de ce chapitre, certaines soustractions seraient peut-être invalidées. Pour éviter ce genre de problème, on peut utiliser la fonction **`difftime()`**, qui permet de manière portable de calculer l'intervalle entre deux instants donnés :

```
double difftime (time_t instant_final, time_t instant_initial);
```

Le type de retour de cette fonction étant un `double`, nous sommes assuré qu'elle pourra gérer sans difficulté d'éventuelles extensions futures du type `time_t`.

Fuseau horaire

Un système Unix en général et Linux en particulier est fondé sur la notion de réseau, de fonctionnement multi-utilisateur et de connexion à l'Internet. Une machine donnée doit pouvoir accepter simultanément des utilisateurs provenant de plusieurs continents, résidant dans des fuseaux horaires totalement différents.

Traditionnellement, les stations Unix utilisent une horloge interne fonctionnant sur la référence TU. Il est donc préférable autant que possible de laisser l'horloge CMOS du PC travailler avec l'heure GMT. L'administrateur de la machine la configure alors pour indiquer dans quel fuseau horaire elle est installée physiquement, et le noyau peut ainsi horodater ses messages, par exemple avec l'heure locale. Malheureusement, pour cause de cohabitation avec d'autres systèmes d'exploitation moins performants, il est souvent nécessaire de laisser l'horloge interne tourner sur la référence locale. Les distributions Linux permettent de gérer ce type de désagrément.

De son côté, un utilisateur peut se connecter sur la machine depuis n'importe quel endroit du monde, et il est normal que le système lui fournisse des informations temporelles adaptées à son environnement. Ceci concerne bien entendu le résultat de la commande `date`, mais également les horodatages des fichiers affichés par `ls`, ou les informations contenues dans l'en-tête des messages électroniques.

Pour simplifier la tâche de l'utilisateur, une seule variable d'environnement sert pour toutes ces opérations : **`TZ`**. Celle-ci doit contenir le nom du fuseau horaire où se trouve l'utilisateur, et toutes les informations de dates et d'heures seront mises à jour automatiquement au moment de l'affichage.

En fait, les fonctions `localtime()`, `mktime()`, `ctime()` ou `strftime()` appellent automatiquement la routine **`tzset()`**, qui sert à initialiser les données correspondant à l'emplacement de l'utilisateur :

```
void tzset(void);
```

Il n'y a normalement pas de raison de faire appel directement à cette routine puisqu'elle est invoquée par toute fonction prenant en compte la position horaire. `tzset()` configure également deux chaînes de caractères globales :

```
char * tzname [2];
```

La chaîne `tzname [0]` contient le nom du fuseau horaire, déterminé depuis la variable d'environnement `TZ`. La chaîne `tzname [1]` comprend le nom de ce fuseau lorsqu'on bascule en heure d'été.

La variable d'environnement TZ peut être remplie avec plusieurs champs successifs, seul le premier étant obligatoire :

- Un nom de fuseau horaire, sur trois caractères au minimum.
- Un décalage qui indique la valeur à ajouter à l'heure TU pour obtenir l'heure locale. La valeur est positive à l'ouest de Greenwich.
- Le nom du fuseau à utiliser pour l'heure d'été.
- Le décalage pour l'heure d'été.
- La date de début de l'heure d'été, indiquée sous l'une des formes suivantes : un J suivi du numéro du jour dans l'année, sans compter le 29 février, même pour les années bissextiles,

un simple numéro de jour, comptant éventuellement le 29 février, ou un M suivi du numéro du mois, puis d'un point, du numéro de la semaine, d'un point, et du numéro du jour de la semaine (le 0 étant le dimanche).

- La date de fin de l'heure d'été.

Les informations concernant les fuseaux horaires préprogrammés sont stockées dans les répertoires `/usr/lib/zoneinfo` ou `/usr/share/zoneinfo` suivant les distributions Linux.

Le programme suivant va afficher quelques exemples de configuration :

exemple_tzname.c :

```
    #include <stdio.h>
    #include <time.h>

    int
main (void)
{
    tzset ();
    fprintf(stdout, "tzname[0] = %s\n", tzname[0]);
    fprintf(stdout, "tzname[1] = %s\n", tzname[1]);
    return EXIT_SUCCESS;
}
```

Nous exécutons le programme qui suit en utilisant d'abord le fuseau horaire de Paris (CET), puis celui de Montréal.

```
$ date
lun jan 17 23:33:05 CET 2000
$ ls -l exemple_tzname.c
-rw-rw-r--   1 ccb      ccb           190 Jan 17 23:22 exemple_tzname.c
$ ./exemple_tzname
tzname[0] = CET
tzname[1] = CEST
$ export TZ=EST
$ date
lun jan 17 17:33:34 EST 2000
$ ls -l exemple_tzname.c
-rw-rw-r--   1 ccb      ccb           190 Jan 17 17:22 exemple_tzname.c
$ ./exemple_tzname
tzname[0] = EST
tzname[1] = EDT
$
```

Nous pouvons faire un essai en inventant notre propre fuseau horaire :

```
$ export TZ=""
$ date
lun jan 17 22:35:49 UTC 2000
$ ./exemple_tzname
tzname[0] = UTC
tzname[1] = UTC
$ export TZ="RIEN -1:12"
$ date
lun jan 17 23:47:59 RIEN  2000
```

```
$ ./exemple_tzname
tzname[0] = RIEN
tzname[1] = RIEN
$
```

Les possibilités de configuration du fuseau horaire utilisateur par utilisateur (et même session par session) offrent des perspectives très intéressantes en ce qui concerne les fonctionnalités de communications internationales, aussi bien pour le courrier électronique que pour les groupes de discussion, en permettant à chacun d'obtenir des informations temporelles intelligibles dans son propre environnement, sans avoir à s'interroger sur la position précise de ses interlocuteurs.

Conclusion

Nous avons examiné en détail l'essentiel des fonctionnalités d'horodatage offertes par Linux et la GlibC.

Insistons encore sur la nécessité de mettre l'accent dès à présent sur la portabilité des programmes qui manipulent des données de type `time_t` et sur le risque d'évolution de celui-ci dans l'avenir.

Le lecteur intéressé pourra trouver des éléments complémentaires sur les notions de calendriers, de dates et d'heures, ainsi que sur les secondes de rattrapage périodique dans les FAQ du groupe Usenet `sci.astro`.

26

Accès aux informations du système

Les informations concernant l'état du système, les groupes et les utilisateurs inscrits, les systèmes de fichiers montés ou les derniers événements survenus sont principalement utilisées dans des applications de configuration (ajout ou suppression d'utilisateurs par exemple) et dans des utilitaires de surveillance du système à destination de l'administrateur.

Dans la plupart des cas, un petit logiciel sert à encadrer l'appel-système ou la fonction de bibliothèque correspondante, et les fonctionnalités de haut niveau sont assurées par un ou plusieurs scripts shell.

Nous allons examiner les fonctions offertes par le noyau et la bibliothèque C pour consulter ou configurer toutes ces données générales sur le paramétrage du système. Comme la plupart de ces données sont conservées dans des fichiers système à peu près similaires, nous verrons que les fonctions présentées dans ce chapitre suivent une inspiration commune.

Groupes et utilisateurs

Les informations concernant les groupes et les utilisateurs inscrits sur le système sont assez largement employées. Bien entendu ceci concerne les utilitaires permettant de gérer la liste des utilisateurs, mais également les applications de communication, la rédaction de courrier électronique, les écrans de connexion graphique au système X-Window, ou tout simplement l'affichage en clair des noms du propriétaire et du groupe d'un fichier.

Fichier des groupes

Les groupes d'utilisateurs sont enregistrés sous Linux dans le fichier `/etc/group`. Ce fichier contient une ligne pour chaque groupe, avec les champs suivants séparés par des deux-points :

- Nom du groupe.
- Mot de passe. Ce champ n'est plus utilisé pour les groupes. On le laisse généralement vide, d'autant qu'il n'est pas défini par Posix.
- GID du groupe, sous forme décimale.
- Liste des noms des utilisateurs appartenant au groupe, séparés par des virgules.

Voici un extrait d'un fichier `/etc/group` :

```
$ cat /etc/group
root::0:root
bin::1:root,bin,daemon
daemon::2:root,bin,daemon
...
nobody::99:
users::100: jennifer, mina, so
...
$
```

Un même nom pouvant être indiqué en troisième argument dans plusieurs lignes du fichier `/etc/group`, cela permet la configuration des groupes supplémentaires de l'utilisateur.

La consultation directe du fichier `/etc/group` est bien entendu déconseillée pour garantir une certaine portabilité du programme vers des systèmes employant d'autres méthodes. La bibliothèque C offre ainsi un certain nombre de fonctions de manipulation des groupes. Une partie de ces fonctions renvoie des données allouées statiquement ; elles disposent à présent d'homologues réentrantes, adaptées à une utilisation dans un contexte multithread.

Pour traiter le contenu des entrées présentes dans le fichier des groupes, on utilise une structure **`group`**, définie dans `<grp.h>` :

Nom	Type	Signification
`gr_name`	`char *`	Nom du groupe
`gr_gid`	`gid_t`	GID du groupe
`gr_mem`	`char **`	Table contenant les noms des utilisateurs, le dernier élément étant un pointeur `NULL`

Les fonctions **`getgrnam()`** et **`getgrgid()`** – ainsi que **`getgrnam_r()`** et **`getgrgid_r()`** – permettent d'obtenir une structure `group` à partir d'un nom ou d'un GID.

```
struct group * getgrnam(const char * nom);
int getgrnam_r (const char * nom, struct group * retour,
                char * buffer, size_t taille_buffer,
                struct group ** pointeur_resultat);
struct group * getgrgid(gid_t gid);
int getgrgid_r (gid_t gid, struct group * retour,
                char * buffer, size_t taille_buffer,
                struct group ** pointeur_resultat);
```

Les fonctions `getgrnam()` et `getgrgid()` renvoient un pointeur sur une zone de mémoire allouée statiquement ou un pointeur `NULL` si aucune entrée n'a été trouvée. Le fonctionnement de `getgrnam_r()` et de `getgrgid_r()` est légèrement plus compliqué du fait qu'il faut fournir un espace pour stocker les chaînes de caractères sur lesquelles la structure regroupe des pointeurs :

- La structure `group` sur laquelle on passe un pointeur en second argument est remplie avec les données lues.
- Le buffer passé en troisième argument, dont la taille est indiquée à la suite, est utilisé pour stocker les chaînes de caractères correspondant aux champs `gr_name` et `gr_mem` de la structure `group`. Si le buffer est trop petit, la fonction échoue et le code d'erreur `ERANGE` est inscrit dans la variable globale `errno`.
- Finalement, le pointeur transmis en dernier argument est rempli avec un pointeur sur la structure `group` passée en deuxième argument, ou avec `NULL` en cas d'échec.

La valeur de retour de `getgrnam_r()` et `getgrgid_r()` est nulle si elles réussissent.

Parfois, on peut être amené à examiner l'ensemble des groupes définis sur la machine, ne serait-ce que dans le cadre d'un utilitaire d'aide à l'administration système. Dans ce cas, les fonctions **`getgrent()`** et **`getgrent_r()`** permettent de lire séquentiellement tous les enregistrements du fichier des groupes.

```
struct group * getgrent (void);
int getgrent_r (struct group * retour,
                char * buffer, size_t taille_buffer,
                struct group ** pointeur_resultat);
```

Ces fonctions utilisent de manière interne un flux correspondant au fichier `/etc/group`. Lors de la première invocation de `getgrent()`, ce flux est ouvert, puis les lignes sont analysées successivement à chaque appel. En fin de fichier, `getgrent()` renvoie un pointeur `NULL`, et `getgrent_r()` une valeur non nulle. Toutefois, on peut avoir besoin de revenir volontairement au début du fichier des groupes. Il existe deux fonctions, **`setgrent()`** et **`endgrent()`**, qui ont un fonctionnement antagoniste, mais avec finalement le même résultat :

```
void setgrent (void);
void endgrent (void);
```

La fonction `setgrent()` ouvre le flux interne utilisé par `getgrent()` et `getgrent_r()`. Si le flux est déjà ouvert, sa position de lecture est ramenée au début. Si `setgrent()` n'est pas appelée explicitement, la première invocation de `getgrent()` le fera automatiquement. La fonction `endgrent()` referme le flux interne. En conséquence, l'une et l'autre de ces fonctions ont pour résultat de faire reprendre la prochaine lecture au début du fichier.

Il est parfois nécessaire de consulter les données se trouvant dans un autre fichier que `/etc/group`. Même si ce genre de situation est rare, on peut la rencontrer par exemple lorsqu'on administre un système distant dont la partition `/etc` est montée par NFS dans un autre emplacement de notre arborescence. Cette situation se présente aussi pour des raisons de sécurité lorsqu'un répertoire est employé comme racine – avec l'appel `chroot()` - pour un processus particulier (comme `/home/ftp`).

Pour cela on dispose des fonctions **`fgetgrent()`** et **`fgetgrent_r()`**, qui permettent de lire depuis un flux qu'on ouvre et qu'on ferme normalement avec `fopen()` et `fclose()`.

```
struct fgetgrent (FILE * flux);
struct fgetgrent_r (FILE * flux, struct group * retour,
                    char * buffer, size_t taille_buffer,
                    struct group ** pointeur_resultat);
```

Ces deux routines fonctionnent exactement comme `getgrent()` et `getgrent_r()`, en se servant simplement du flux transmis en premier argument plutôt que de le gérer elles-mêmes.

On notera l'existence d'une fonction de bibliothèque nommée **initgroups()**, qui sert à consulter le fichier des groupes et à initialiser la liste des groupes supplémentaires d'un utilisateur donné.

```
int initgroups (const char * nom, gid_t gid);
```

Cette routine est privilégiée car elle invoque `setgroups()`, que nous avons étudiée dans le chapitre 2, et qui réclame la capacité `CAP_SETGID`. Cette fonction est appelée lors de la connexion d'un utilisateur par l'utilitaire `/bin/login`, ainsi que par `/bin/su`. On ne l'invoque normalement pas dans une application courante.

Fichier des utilisateurs

Comme le fichier des groupes, celui des utilisateurs est stocké dans le répertoire `/etc`. Typiquement, il s'agit du fichier `/etc/passwd`. Ce dernier est accessible en lecture pour tous les utilisateurs du système (ceci est nécessaire par exemple pour trouver le nom réel d'un utilisateur à partir de son UID). Toutefois, le mot de passe y apparaît de manière cryptée, comme nous en avons parlé dans le chapitre 16. L'évolution des processeurs rend à présent possible la recherche de mots de passe par force brute, en cryptant successivement tout le dictionnaire pour découvrir la chaîne chiffrée correspondant à celle du fichier `/etc/passwd`. Pour cette raison, la plupart des systèmes Linux utilisent à présent la technique des *shadow passwords*. Le mot de passe crypté n'est plus stocké dans `/etc/passwd` mais dans un autre fichier, comme `/etc/shadow`, accessible en lecture uniquement par un processus ayant un UID nul.

Le fichier contient une ligne pour chaque utilisateur, avec un certain nombre de paramètres. Pour manipuler ces enregistrements, on utilise la structure **passwd**, définie dans `<pwd.h>` :

Nom	Type	Signification
`pw_name`	`char *`	Nom de l'utilisateur, tel qu'il est employé pour la connexion.
`pw_passwd`	`char *`	Mot de passe crypté.
`pw_uid`	`uid_t`	UID de l'utilisateur.
`pw_gid`	`gid_t`	GID principal de l'utilisateur.
`pw_gecos`	`char *`	Commentaires sur l'utilisateur.
`pw_dir`	`char *`	Répertoire personnel de l'utilisateur.
`pw_shell`	`char *`	Le shell employé lors de la connexion de l'utilisateur.

Nous pouvons relever quelques points :

- Le membre `pw_passwd` n'est pas significatif sur les systèmes employant les *shadow passwords*. Si vous désirez écrire une application qui s'assure de l'identité d'un utilisateur en vérifiant son mot de passe, il faut que celle-ci puisse accéder au fichier `/etc/shadow` plutôt que `/etc/passwd`, et qu'elle puisse y lire les enregistrements.
- Le champ `pw_gid` correspond au groupe principal de l'utilisateur. L'accès aux GID de ses groupes supplémentaires est possible avec la fonction `getgroups()`, que nous avons vue dans le chapitre 2.
- Le champ `pw_gecos` peut contenir des informations plus ou moins pertinentes suivant les habitudes d'administration sur le système. Dans certains cas, il est vide ou ne comprend

que le nom complet de l'utilisateur. À l'opposé, on peut rencontrer des systèmes où ce champ est lui-même scindé en plusieurs sous-enregistrements séparés par des virgules comportant adresse, numéro de téléphone, numéro de fax, etc.

- Le répertoire personnel de l'utilisateur indiqué dans `pw_dir` est normalement accessible en lecture, parcours et écriture par l'utilisateur. Pour certains comptes particuliers (connexion PPP entrante par exemple), il peut s'agir d'un répertoire uniquement accessible en parcours, voire de la racine du système de fichiers.
- De même, le shell employé pour la connexion, indiqué dans le champ `pw_shell`, peut dans certains cas particuliers être un programme spécial (`/sbin/shutdown` par exemple).

L'accès à la structure `passwd` s'obtient par le biais de fonctions qui ressemblent largement à celles que nous avons rencontrées pour les groupes d'utilisateurs. Pour avoir l'entrée correspondant à un utilisateur donné, **`getpwuid()`** et **`getpwnam()`** permettent des recherches respectivement sur l'UID ou le nom de connexion. Leurs versions réentrantes, **`getpwuid_r()`** et **`getpwnam_r()`**, fonctionnent sur le même modèle que celui qui a été décrit pour `getgrnam_r()`.

```
struct passwd * getpwuid (uid_t uid);
int getpwuid_r (uid_t uid, struct passwd * retour,
                char * buffer, size_t taille_buffer,
                struct passwd ** pointeur_resultat);
struct passwd * getpwnam (const char * nom);
int getpwnam_r (const char * nom, struct passwd * retour,
                char * buffer, size_t taille_buffer,
                struct passwd ** pointeur_resultat);
```

Lorsqu'on veut balayer tout le fichier des utilisateurs, on peut utiliser la fonction **`getpwent()`** ou sa cousine réentrante, **`getpwent_r()`**.

```
struct passwd * getpwent (void);
int getpwent_r (struct passwd * retour,
                char * buffer, size_t taille_buffer,
                struct passwd ** pointeur_resultat);
```

Pour réinitialiser la lecture séquentielle du fichier des utilisateurs, on a le choix entre **`setpwent()`**, qui ouvre le flux interne, ou **`endpwent()`**, qui le ferme.

```
void setpwent (void);
void endpwent (void);
```

Quand on désire travailler avec un autre fichier que `/etc/passwd` (notamment `/etc/shadow` ou `/home/ftp/etc/passwd`), on emploie les fonctions **`fgetpwent()`** ou **`fgetpwent_r()`** en leur transmettant le pointeur de flux correspondant au fichier déjà ouvert.

```
struct passwd * fgetpwent (FILE * flux);
int fgetpwent_r (FILE * flux, struct passwd * retour,
                char * buffer, size_t taille_buffer,
                struct passwd ** pointeur_resultat);
```

La fonction **`putpwent()`** permet de créer un enregistrement dans un fichier d'utilisateurs dont on lui fournit un pointeur de flux. L'emploi de cette routine est déconseillé car elle ajoute simplement l'enregistrement, sans vérifier s'il existait auparavant. Finalement, les applications devant modifier le fichier des mots de passe (comme `/usr/bin/passwd` bien entendu,

mais aussi des utilitaires comme /usr/bin/chsh) devront être installées Set-UID *root* et modifieront directement le fichier.

```
int putpwent (const struct passwd * passwd, FILE * flux);
```

Une fonctionnalité importante du fichier des mots de passe concerne l'obtention du nom de connexion de l'utilisateur ayant lancé le processus courant. En réalité, les fonctions **getlogin()** et **getlogin_r()** donnent accès au nom de l'utilisateur connecté sur le terminal de contrôle du processus. Ces routines, à la différence des précédentes, sont déclarées dans <unistd.h> :

```
char * getlogin (void);
int getlogin_r (char * buffer, size_t taille_buffer);
```

La fonction **cuserid()** fournit le nom de l'utilisateur correspondant à l'UID effectif du processus appelant. Elle est déclarée dans <stdio.h> :

```
char * cuserid (char * nom);
```

Si le buffer transmis en argument n'est pas NULL, il doit faire au moins **L_cuserid** octets de long, et le nom y est stocké. Sinon, la fonction renvoie un pointeur sur une zone de mémoire allouée statiquement.

Attention

Cette routine n'est pas portable ; elle est implémentée différemment sur d'autres Unix. Elle est considérée désormais comme obsolète.

Fichier des interpréteurs shell

Il existe normalement un fichier /etc/shells, qui contient, ligne par ligne, la liste des interpréteurs de commandes disponibles. En voici un exemple sur une distribution Red Hat :

```
$ cat /etc/shells
/bin/bash
/bin/sh
/bin/ash
/bin/bsh
/bin/tcsh
/bin/csh
/bin/ksh
/bin/zsh
$
```

Cette liste est employée principalement par l'utilitaire /usr/bin/chsh et par les programmes d'aide à l'administration système pour ajouter un utilisateur. Pour lire le contenu de ce fichier, les fonctions **getusershell()**, **setusershell()** et **endusershell()** sont déclarées dans <unistd.h> :

```
char * getusershell (void);
void   setusershell (void);
void   endusershell (void);
```

Comme nous l'avons déjà observé avec le fichier des groupes et celui des utilisateurs, la fonction `getusershell()` permet de lire le fichier des shells séquentiellement. `setusershell()` et `endusershell()`ramènent quant à elles la position de lecture au début.

Un utilisateur n'est pas forcé de choisir son shell de connexion dans cette liste. Par exemple, dans le cas d'un nom utilisé pour mettre en place une connexion PPP entrante, le programme de connexion indiqué dans le fichier des utilisateurs pourra être `/usr/sbin/pppd`. On notera également que si le fichier des shells n'est pas accessible, `getusershell()` se comporte comme si celui-ci contenait les lignes `/bin/sh` et `/bin/csh`.

Nom d'hôte et de domaine

Nom d'hôte

Le nom d'hôte est principalement employé pour identifier le système lors d'un dialogue avec un utilisateur humain. Ce nom ne sert généralement pas lors des communications entre ordinateurs, où on utilise plutôt des identifications numériques comme l'adresse IP ou l'adresse MAC. Pour obtenir le nom de la machine sur laquelle une application se déroule, on emploie la fonction **`gethostname()`**. La routine privilégiée **`sethostname()`** sert à configurer le nom d'hôte. Elle est généralement invoquée une seule fois dans un script de démarrage par le biais de l'utilitaire `/bin/hostname` qui lui sert d'interface.

```
int gethostname (char * buffer, size_t taille);
int sethostname (char * buffer, size_t taille);
```

La taille du buffer contenant le nom est transmise en seconde position. Si le buffer est trop petit pour recevoir la chaîne de caractères, `gethostname()` échoue avec l'erreur `ENAMETOOLONG`.

Le nom d'hôte qu'on manipule avec `gethostname()` ou `sethostname()` doit être complet, c'est-à-dire qu'il doit contenir le domaine en entier. En voici un exemple d'utilisation :

exemple_gethostname.c :

```
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    char * buffer = NULL;
    size_t taille = 8;

    buffer = malloc(taille);

    while (gethostname(buffer, taille) != 0) {
        if (errno != ENAMETOOLONG) {
            perror("gethostname");
            exit(EXIT_FAILURE);
        }
        taille += 8;
        buffer = realloc(buffer, taille);
```

```
    }
    fprintf(stdout, "%s\n", buffer);
    free(buffer);
    return EXIT_SUCCESS;
}
```

Nous pouvons vérifier que le nom de la machine est bien complet :

```
$ ./exemple_gethostname
venux.logilin.fr
$
```

Lorsqu'on utilise l'appel-système `sethostname()`, on signale simplement en second argument la longueur du nom, telle qu'elle est fournie par `strlen()`. Si ce nom est trop long, la fonction renvoie l'erreur `EINVAL` dans `errno`. Pour être autorisé, le changement de nom d'hôte doit être réalisé par un processus ayant la capacité `CAP_SYS_ADMIN`.

Nom de domaine

Parallèlement, il existe deux appels-système, **`getdomainname()`** et **`setdomainname()`**, qui permettent d'obtenir et de configurer le nom du domaine auquel appartient la machine.

```
int getdomainname (char * buffer, size_t taille);
int setdomainname (char * buffer, size_t taille);
```

En fait, `getdomainname()` n'est plus un appel-système sous Linux, il a été remplacé par une fonction de bibliothèque qui utilise l'appel `uname()`. On notera que `getdomainname()` renvoie toujours une chaîne vide si on ne prend pas le système NIS pour déterminer le domaine. Sur la plupart des stations Linux autonomes, cette fonction n'est donc pas utile.

Identifiant d'hôte

Pour essayer d'identifier une machine de manière unique, la bibliothèque C définit deux fonctions, **`gethostid()`** et **`sethostid()`**. La seconde est une fonction privilégiée qui enregistre l'identifiant qu'on lui transmet dans un fichier (généralement `/var/adm/hostid`). Lorsque `gethostid()` est invoquée sans qu'on ait appelé `sethostid()` auparavant, elle utilise l'adresse IP de la première interface réseau de la machine. Si cette opération se révèle impossible, elle renvoie 0.

```
long int gethostid (void);
int sethostid (long int identifiant);
```

L'identifiant peut, sur certains systèmes autres que Linux, être construit directement à partir de l'adresse MAC de l'interface réseau.

Informations sur le noyau

Identification du noyau

Il peut être utile dans une application d'identifier la version du noyau en cours d'exécution (par exemple pour tirer parti de certaines nouvelles fonctionnalités ou pour éviter un bogue présent dans une ancienne version). L'appel-système **`uname()`** permet d'obtenir plusieurs renseignements sur le système.

```
int uname (struct utsname * utsname);
```

Les informations sont stockées dans une structure `utsname` définie dans `<sys/utsname.h>`, et dont tous les membres sont du type `char *` :

Nom	Signification
`sysname`	Nom du système d'exploitation. Pour nous, « `Linux` ».
`nodename`	Nom complet de la machine, comme avec `gethostname()`.
`release`	Numéro de version du noyau.
`version`	Numéro de révision du noyau au sein de la version courante.
`machine`	Type de machine. Il s'agit du nom du processeur suivi de celui du fabricant de l'ordinateur. Ce dernier nom n'est pas toujours disponible ; on obtient souvent quelque chose comme « `i686-unknown` ».
`domainname`	Nom du domaine auquel appartient la machine, comme dans `getdomainname()`. Pour que ce champ soit disponible, il faut définir la constante symbolique `_GNU_SOURCE` avant l'inclusion de l'en-tête `<sys/utsname.h>`.

Voici un programme simple permettant de visualiser les différents champs :

exemple_uname.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <sys/utsname.h>

    int
main (void)
{
    struct utsname    utsname;
    uname(& utsname);
    fprintf(stdout, " sysname  = %s \n nodename = %s \n"
                    " release  = %s \n version  = %s \n"
                    " machine  = %s  \n domaine  = %s \n",
                utsname.sysname,
                utsname.nodename,
                utsname.release,
                utsname.version,
                utsname.machine,
                utsname.domainname);
    return EXIT_SUCCESS;
}
```

Et voici un exemple d'exécution :

```
$ ./exemple_uname
 sysname  = Linux
 nodename = venux.logilin.fr
 release  = 2.6.9
 version  = #2 Tue Jan 4 13:05:35 CET 2005
 machine  = i686
 domaine  =
$
```

Informations sur l'état du noyau

Il est rare dans un cadre applicatif d'avoir réellement besoin d'obtenir des renseignements pointus sur l'état du système. Toutefois, cela est possible sous Linux à l'aide de l'appel-système **sysinfo()**, déclaré dans <sys/sysinfo.h> :

```
int sysinfo (struct sysinfo * info);
```

Cet appel-système n'est absolument pas portable sur d'autres systèmes que Linux. La structure **sysinfo** qu'il remplit est définie dans <linux/kernel.h> avec les membres suivants :

Nom	Type	Signification
uptime	Long	Nombre de secondes écoulées depuis le boot de la machine
loads	unsigned long [3]	Charge système durant les 1, 5 et 10 dernières minutes, 65 535 correspondant à 100 %
freeram	unsigned long	Mémoire libre, exprimée en octets
sharedram	unsigned long	Mémoire partagée entre plusieurs processus
bufferram	unsigned long	Mémoire utilisée pour les buffers du noyau
totalswap	unsigned long	Taille totale du périphérique de swap, en octets
freeswap	unsigned long	Quantité disponible sur le périphérique de swap
procs	unsigned short	Nombre de processus en cours d'exécution

Le programme suivant met en œuvre l'appel-système sysinfo(), à la manière des utilitaires /usr/bin/uptime ou /usr/bin/top.

exemple_sysinfo.c :

```
    #include <stdio.h>
    #include <sys/sysinfo.h>
    #include <linux/kernel.h>

    int
main (void)
{
    struct sysinfo info;
    if (sysinfo(& info) != 0) {
        perror("sysinfo");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Nb secondes depuis boot    : %ld \n",
        info.uptime);
    fprintf(stdout, "Charge système depuis 1 mn : %.2f%%\n",
        info.loads[0] / 655.36);
    fprintf(stdout, "                     5 mn : %.2f%%\n",
        info.loads[1] / 655.36);
    fprintf(sdout, "                    10 mn : %.2f%%\n",
        info.loads[2] / 655.36);
    fprintf(stdout, "Mémoire disponible         : %ld Mo\n",
        info.freeram >> 20);
```

```
    fprintf(stdout, "Mémoire partagée           : %ld Mo\n",
        info.sharedram >> 20);
    fprintf(stdout, "Mémoire dans buffers       : %ld Mo\n",
        info.bufferram >> 20);
    fprintf(stdout, "Espace de swap total       : %ld Mo\n",
        info.totalswap >> 20);
    fprintf(stdout, "Espace de swap libre       : %ld Mo\n",
        info.freeswap  >> 20);
    fprintf(stdout, "Nb processus en cours      : %d  \n",
        info.procs);
    return EXIT_SUCCESS;
}
```

Rappelons que cet appel-système est spécifique à Linux et qu'il ne sera pas disponible sur d'autres architectures.

```
$ ./exemple_sysinfo
Nb secondes depuis boot    : 170191
Charge système depuis 1 mn : 1.20%
                      5 mn : 0.05%
                     10 mn : 0.01%
Mémoire disponible         : 5 Mo
Mémoire partagée           : 0 Mo
Mémoire dans buffers       : 138 Mo
Espace de swap total       : 509 Mo
Espace de swap libre       : 509 Mo
Nb processus en cours      : 59
$
```

Système de fichiers

Un système Linux normal gère au minimum deux partitions physiques : l'une constitue la racine de l'arborescence du système de fichiers et l'autre est utilisée comme périphérique de swap. On emploie également d'autres systèmes de fichiers pour accéder aux périphériques amovibles, comme les lecteurs de CD-Rom ou les disquettes. Il peut aussi être intéressant de se servir de plusieurs partitions physiques différentes sur le disque, qu'on monte à différents endroits du système de fichiers. Ceci permet par exemple d'utiliser un espace disque limité et figé pour les répertoires ne devant pas évoluer sensiblement (`/`, `/bin`, `/etc`, `/dev`, `/usr`), et de laisser le reste de la capacité disponible pour les zones susceptibles de subir des modifications importantes (`/home`, `/tmp`). On peut également utiliser un découpage du disque en plusieurs partitions pour simplifier les travaux de sauvegardes systématiques en isolant les données modifiées fréquemment de celles qui n'évoluent pas une fois le système installé. Lorsqu'on administre un parc de plusieurs stations Unix, il est très important qu'un utilisateur puisse disposer de son environnement de travail personnel quelle que soit la machine devant laquelle il s'assoit. On organise alors une distribution des répertoires personnels `/home/xxx` sur l'ensemble des machines, avec un montage au travers du réseau par le protocole NFS.

Enfin, on peut être amené à personnaliser le partitionnement pour des besoins spécifiques. À titre d'exemple, je peux citer un cas où je devais installer des stations Linux dans un environnement de production assez périlleux, sujet à de fréquents problèmes de distribution électrique et à des manipulations pour le moins maladroites. J'ai décidé d'employer un montage en lecture seule de tous les répertoires système (`/usr`, `/etc`, …). Les données propres à l'utilisateur ainsi

que les fichiers de configuration susceptibles de changer se trouvaient dans des répertoires (/home et /usr/local) montés par NFS depuis un serveur administré par une équipe de maintenance opérationnelle. Enfin, une partition reformatée à chaque démarrage de la machine regroupait les fichiers à faible durée de vie dans /tmp. L'avantage d'une telle organisation était de permettre un arrêt brutal de la machine sans risque de perte de données.

Caractéristiques des systèmes de fichiers

Pour simplifier le travail de l'administrateur – afin d'autoriser l'insertion de périphériques amovibles par n'importe quel utilisateur et permettre le montage automatique de certaines partitions au démarrage de la machine –, on établit la liste des systèmes de fichiers disponibles et on la stocke dans /etc/fstab. Il sera alors possible de rattacher une partition au système de fichiers simplement en invoquant une commande du type :

```
$ mount /mnt/cdrom
$ mount /mnt/dos
$ mount /mnt/floppy
```

De plus, les gestionnaires graphiques de fichiers sauront monter ou démonter les nœuds correspondant en consultant cette table. Chaque ligne du fichier /etc/fstab contient les champs suivants, séparés par des tabulations ou par des espaces.

- Nom du fichier spécial en mode bloc représentant le périphérique (par exemple /dev/hda1 pour une partition disque IDE, /dev/fd0 pour le premier lecteur de disquette, etc.). Pour les pseudo-systèmes de fichiers comme /proc, on indique souvent le mot-clé none. Les répertoires provenant d'un serveur NFS sont mentionnés en signalant le nom du serveur (ou son adresse IP), suivi d'un deux-points et du chemin d'accès au répertoire. Par exemple pingouin:/home/tux.
- Le point de montage du système de fichiers dans l'arborescence générale (par exemple / pour la partition racine, /mnt/cdrom ou /mnt/floppy pour des périphériques amovibles, /home/users pour un répertoire normal). Pour les partitions de swap, on utilise le mot-clé none.
- Le type du système de fichiers. Le noyau Linux reconnaît couramment les systèmes minix (héritage historique des premiers noyaux), ext (obsolète, à oublier), ext3 (le standard Linux actuel), ext2 (son prédécesseur), iso9660 (pour les CD-Rom), udf (pour les DVD), swap, nfs, msdos, ntfs, et vfat (pour accéder aux partitions Dos ou Windows). On peut mentionner les pseudo systèmes de fichiers proc (informations du noyau) et devpts (pseudo terminaux Unix 98 comme nous les étudierons dans le chapitre 33). Il existe de nombreux autres systèmes, moins utilisés ou expérimentaux, qui sont généralement disponibles sous forme de modules du noyau.
- Des options concernant le montage du système de fichiers. Il existe des options générales, comme noauto qui empêche le montage automatique au démarrage (utile pour les périphériques amovibles), user qui autorise n'importe quel utilisateur à monter le système de fichiers, ro qui demande un montage en lecture seule, ou mand qui permet les verrouillages stricts, comme nous l'avons vu au chapitre 19. Il y a aussi des options spécifiques pour chaque type de système. On consultera au besoin les pages de manuel mount(8), fstab(5) et nfs(5) pour avoir des détails supplémentaires.
- La fréquence des sauvegardes de la partition par l'utilitaire dump. Cette option n'est généralement pas utilisée, on la remplace par un zéro.

- L'ordre de vérification des systèmes de fichiers au démarrage de la machine. Si ce champ est absent ou nul, la partition n'est pas vérifiée. Sinon, le programme `fsck` traite les systèmes de fichiers séquentiellement dans l'ordre indiqué. Normalement, la partition racine doit être configurée avec la valeur 1, et les autres avec la valeur 2. Si les partitions sont gérées par des contrôleurs de disque distincts, elles sont vérifiées en parallèle.

Voici un extrait d'un tel fichier :

```
$ cat /etc/fstab
/dev/hda5       /              ext3        defaults,mand      0   1
/dev/hda6       swap           swap        defaults           0   0
/dev/fd0        /mnt/floppy    vfat        noauto,user        0   0
/dev/hdc        /mnt/cdrom     udf,iso9660 noauto,ro,user     0   0
/dev/hda1       /mnt/dos       vfat        noauto,user        0   0
none            /dev/pts       devpts      gid=5,mode=620     0   0
none            /proc          proc        defaults           0   0
        ...
$
```

Naturellement, il est toujours possible d'invoquer directement la commande `mount` avec toutes les options en ligne de commande, mais il est beaucoup plus agréable de n'avoir à saisir que `mount /mnt/cdrom` par exemple. Il faut donc accorder une grande importance à la rédaction du fichier `/etc/fstab`, d'autant que cette tâche d'administration n'a lieu qu'une seule fois, lors de l'installation du système (ou en cas d'ajout d'un nouveau périphérique). La bibliothèque C propose un ensemble de fonctions permettant de consulter ce fichier. Les routines **`setfsent()`** et **`endfsent()`** fonctionnent comme d'habitude en ouvrant ou en fermant le fichier `/etc/fstab`, ce qui a donc pour conséquence de faire reprendre la lecture suivante au début.

```
int setfsent (void);
void endfsent (void);
```

Pour manipuler les enregistrements, une structure `fstab` est définie dans `<fstab.h>` :

Nom	Type	Signification
`fs_spec`	`char *`	Nom du fichier spécial représentant le périphérique concerné. Peut également être un nom d'hôte suivi d'un chemin d'accès pour les montages NFS.
`fs_file`	`char *`	Point de montage dans l'arborescence du système de fichiers.
`fs_vfstype`	`char *`	Type de système de fichiers.
`fs_mntops`	`char *`	Options de montage, globales ou spécifiques au type de système de fichiers employé.
`fs_type`	`char *`	Mode d'accès à la partition montée (voir plus bas).
`fs_freq`	`int`	Période (en jours) entre deux sauvegardes (souvent inutilisé).
`fs_passno`	`int`	Ordre de vérification de la partition au démarrage.

Le champ `fs_type` peut contenir l'une des chaînes de caractères décrites dans le tableau suivant.

Attention

Il s'agit bien de chaînes de caractères et non de constantes symboliques. Il faut donc les examiner avec `strcmp()`.

Chaîne	Signification
FSTAB_RW	Partition à monter en lecture et écriture (par exemple /tmp)
FSTAB_RQ	Partition à monter en lecture et écriture avec un système de comptabilité par quotas (par exemple /home)
FSTAB_RO	Partition à monter en lecture seule (par exemple /usr)
FSTAB_SW	Partition de swap
FSTAB_XX	Partition ignorée

La fonction **getfsent()** renvoie l'entrée suivante du fichier, **getfsspec()** recherche l'entrée dont le champ fs_spec (nom du fichier spécial de périphérique) correspond à la chaîne transmise en argument. La fonction **getfsfile()** retourne quant à elle l'entrée dont le membre fs_file (point de montage) correspond à son argument.

```
struct fstab * getfsent (void);
struct fstab * getfsspec (const char * nom);
struct fstab * getfsfile (const char * nom);
```

Ces fonctions renvoient un pointeur sur une structure stockée dans une zone de mémoire statique, ou NULL en cas d'échec.

Pour connaître l'état des systèmes de fichiers actuellement montés, un processus peut examiner le pseudo-fichier /proc/mounts mis à jour par le noyau. Toutefois, ceci n'est pas portable, et le format de ce fichier peut évoluer dans des versions futures de Linux. Pour simplifier cette tâche, les utilitaires mount et umount mettent à jour une table, stockée généralement dans le fichier /etc/mtab, avec un format ressemblant à celui de /etc/fstab mais ne contenant que les partitions montées.

Pour analyser ces données, on utilise une structure mntent définie dans <mntent.h>, dont les membres sont mieux nommés que ceux de fstab :

Nom	Type	Signification
mnt_fsname	char *	Nom du fichier spécial de périphérique (équivalent à fs_spec)
mnt_dir	char *	Point de montage (équivalent à fs_file)
mnt_type	char *	Le type du système de fichiers (équivalent à fs_vfstype)
mnt_opts	char *	Options utilisées durant le montage (équivalent à fs_mntops)
mnt_freq	int	Fréquence de sauvegarde (équivalent à fs_freq)
mnt_passno	int	Ordre de vérification (équivalent à fs_passno)

Les fonctions **setmntent()** et **endmntent()** permettent d'ouvrir et de fermer un flux en fournissant le nom du fichier. Ces routines rendent possible la manipulation de /etc/mtab mais également d'autres fichiers ayant le même format, comme /etc/fstab ou /proc/mounts.

```
FILE * setmntent (const char * nom, const char * mode);
int endmntent (FILE * fichier);
```

Les arguments de setmntent() sont identiques à ceux de fopen().

Attention
Le pointeur de flux renvoyé par `setmntent()` doit être refermé à l'aide de `endmntent()` et surtout pas avec `fclose()`.

La lecture peut se faire à l'aide de **`getmntent()`**, qui renvoie un pointeur sur une zone statique, ou avec **`getmntent_r()`**, réentrante, qui utilise des arguments plus compliqués, comme nous l'avons déjà observé avec `getgrnam()`. Pour éviter les problèmes de taille du buffer, il suffit que celui-ci soit suffisamment grand pour contenir la plus longue ligne du fichier. Les lignes d'un fichier `/etc/mtab` ne dépassent généralement pas 80 caractères.

```
struct mntent * getmntent (FILE * fichier);
struct mntent * getmntent_r (FILE * fichier, struct mntent * retour,
                             char * buffer, int taille_buffer);
```

Le programme suivant va utiliser `getmntent_r()` – bien que `getmntent()` aurait largement suffi dans ce contexte monothread – afin de consulter le fichier dont le nom est fourni en argument.

exemple_mtab.c :

```
    #include <stdio.h>
    #include <mntent.h>

    int
main (int argc, char * argv[])
{
    struct mntent mtab;
    char buffer[256];
    FILE * file;

    if (argc != 2) {
        fprintf(stderr, "%s <fichier mtab> \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((file = setmntent(argv[1], "r")) == NULL) {
        perror("setmntent");
        exit(EXIT_FAILURE);
    }
    while (1) {
        if (getmntent_r(file, & mtab, buffer, 256) == NULL)
            break;
        fprintf(stdout, "fsname = %s \n dir = %s\n type = %s \n"
                        " opts = %s \n freq = %d \n passno = %d \n",
                         mtab.mnt_fsname, mtab.mnt_dir,
                         mtab.mnt_type,   mtab.mnt_opts,
                         mtab.mnt_freq,   mtab.mnt_passno);
    }
    endmntent(file);
    return EXIT_SUCCESS;
}
```

On peut utiliser ce programme sur /etc/fstab, /etc/mtab, /proc/mounts…

```
$ ./exemple_mtab /proc/mounts
fsname = /dev/root
 dir = /
 type = ext3
 opts = rw,mand
 freq = 0
 passno = 0
fsname = /proc
 dir = /proc
 type = proc
 opts = rw
 freq = 0
 passno = 0
fsname = none
 dir = /dev/pts
 type = devpts
 opts = rw
 freq = 0
 passno = 0
$
```

Il est aussi possible d'ajouter une nouvelle entrée dans un fichier, à l'aide de **addmntent()**, qui permet d'écrire une application incorporant des routines de montage et de démontage de systèmes de fichiers tout en restant compatible avec l'utilitaire mount. On peut également employer ces routines pour créer un éditeur de fichier /etc/fstab. Comme il n'existe pas de routine spécialisée, si on veut supprimer une ou plusieurs lignes, il faut recopier le fichier entrée par entrée, en sautant celles qu'on veut éliminer, et utiliser rename() pour remplacer le fichier original. Naturellement, l'ajout d'enregistrements dans un fichier nécessite l'ouverture en mode r+.

```
int addmntent (FILE * fichier, const struct mntent * mntent);
```

Pour analyser le champ mnt_opts de la structure mntent, il est conseillé d'utiliser la routine getsubopt() que nous avons étudiée dans le chapitre 3. Toutefois, lorsqu'on désire simplement vérifier la présence d'une option bien déterminée dans une entrée, on peut plutôt choisir la fonction **hasmntopt()**.

```
char * hasmntopt (const struct mntent * mntent, const char * option);
```

Si l'option indiquée en second argument se trouve dans le champ mnt_opts de la structure passée en première position, cette fonction renvoie un pointeur sur le premier caractère de cette option (dans la chaîne mnt_opts). Sinon, elle transmet un pointeur NULL. Ainsi le programme suivant recherche les partitions mentionnées dans /etc/fstab qui possèdent l'attribut mand autorisant un verrouillage strict des fichiers.

exemple_hasmntopt.c :

```
    #include <stdio.h>
    #include <mntent.h>

    int
main (void)
```

```
{
    FILE * fichier;
    struct mntent * mntent;

    fichier = setmntent("/etc/fstab", "r");
    if (fichier == NULL)
        exit(EXIT_FAILURE);
    while (1) {
        mntent = getmntent(fichier);
        if (mntent == NULL)
            break;
        if (hasmntopt(mntent, "mand") != NULL)
            fprintf(stdout, "%s (%s)\n",
                    mntent->mnt_fsname,
                    mntent->mnt_dir);
    }
    endmntent(fichier);
    return EXIT_SUCCESS;
}
```

On retrouve alors la partition racine de l'exemple précédent :

```
$ ./exemple_hasmntopt
/dev/hda5 (/)
$
```

Informations sur un système de fichiers

Pour obtenir des informations concernant un système de fichiers particulier, s'il est monté, il est possible d'employer les appels-système **statfs()** ou **fstatfs()**. Le premier fournit des renseignements sur le système contenant le fichier (ou le répertoire) dont le nom est passé en argument. Le second appel-système utilise un descripteur de fichier, qui doit donc avoir été préalablement ouvert.

```
int statfs  (const char * fichier, struct statfs * statfs);
int fstatfs (int descripteur, struct statfs * statfs);
```

Les informations sont transmises dans une structure `statfs`, définie dans `<sys/statfs.h>` avec au minimum les membres suivants :

Nom	Type	Signification
`f_type`	`int`	Type de système de fichiers
`f_bsize`	`int`	Taille de bloc
`f_blocks`	`long`	Nombre total de blocs
`f_bfree`	`long`	Nombre de blocs libres
`f_bavail`	`long`	Nombre de blocs vraiment disponibles
`f_files`	`long`	Nombre d'i-nœuds
`f_ffree`	`long`	Nombre d'i-nœuds libres
`f_fsid`	`fsid_t`	Identifiant du système de fichiers (peu utilisé actuellement)
`f_namelen`	`int`	Longueur maximale des noms de fichiers

Le type de système de fichiers est indiqué sous forme numérique. Il existe une constante symbolique pour chaque système connu par le noyau. Ces constantes sont définies dans les fichiers d'en-tête `<linux/<XXX>_fs.h>`, où `<XXX>` correspond au nom du système de fichiers. La constante symbolique est alors `<XXX>_SUPER_MAGIC`. Par exemple, le système de fichiers `ext2` est représenté par la constante symbolique `EXT2_SUPER_MAGIC`, définie dans `<linux/ext2_fs.h>`, et prend la valeur `0xEF53`. Une application n'a normalement pas besoin de connaître ces valeurs.

Nous pouvons utiliser cet appel-système pour obtenir des informations statistiques sur l'utilisation du système auquel appartient le fichier dont le nom est passé en argument.

exemple_statfs.c :

```
    #include <stdio.h>
    #include <sys/vfs.h>

    int
main (int argc, char * argv[])
{
    struct statfs   etat;
    int             i;
    for (i = 1; i < argc; i ++) {
        if (statfs(argv[i], & etat) < 0) {
            perror(argv[i]);
            continue;
        }
        fprintf(stdout, "%s : 1 bloc = %ld octets \n"
                 "    total %ld blocs \n"
                 "    libre %ld blocs \n"
                 "    disponible %ld blocs \n",
                argv[i],  etat.f_bsize,
                argv[i],  etat.f_bsize,
                etat.f_blocks, etat.f_bfree, etat.f_bavail);
    }
    return EXIT_SUCCESS;
}
```

L'exécution donne un résultat comparable à la commande `/bin/df`, qui affiche les mêmes statistiques pour tous les systèmes de fichiers en employant `getmntent()` sur `/etc/mtab`, comme nous l'avons déjà fait.

```
$ ./exemple_statfs /etc/
/etc/ : 1 bloc = 4096 octets
    total 2579465 blocs
    libre 1733633 blocs
    disponible 1602603 blocs
$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5             10317860   3383328   6410412  35% /
$
```

Les appels-système `statfs()` et `fstatfs()` ne sont pas définis par SUSv3 et ils ne sont que moyennement portables. Si cela pose un problème, on peut s'inspirer des sources de l'application `/bin/df` qui emploie des appels-système différents suivant les machines, pour obtenir

finalement le même résultat. Notons également au passage la persistance d'un appel-système **ustat()** provenant de Système V, offrant un sous-ensemble des informations fournies par statfs(), et donc quasi obsolète de nos jours.

Montage et démontage des partitions

Nous ne détaillerons pas les mécanismes utilisés pour monter ou démonter les partitions, car ils sont susceptibles de changer en fonction des versions du noyau. Précisons simplement rapidement la structure des appels-système **mount()** et **umount()** qui remplissent ce rôle. Ils sont déclarés dans <sys/mount.h> :

```
int mount (const char * fichier_special,
           const char * point_montage, const char * type_systeme,
           unsigned long attribut, const void * options);
int umount (cont char * nom);
```

Les arguments principaux de mount() ressemblent aux champs du fichier /etc/fstab. Un argument supplémentaire est présent pour préciser des attributs de montage. Il doit être rempli partiellement avec une valeur magique (0xC0ED) et complété par des valeurs spéciales définies dans le fichier d'en-tête <linux/fs.h>, dépendant donc du noyau.

Ces appels-système sont privilégiés, nécessitant la capacité CAP_SYS_ADMIN. L'utilitaire /bin/mount est donc installé Set-UID *root*, et il gère lui-même les options user ou nouser qui autorisent ou interdisent le montage d'un système par n'importe quel utilisateur. Ces options sont internes à l'application et ne concernent pas l'appel-système.

Si on désire créer un logiciel qui permette à l'utilisateur de monter ou de démonter des partitions décrites dans /etc/fstab (comme un gestionnaire graphique de fichiers), on emploiera donc de préférence les invocations suivantes :

```
execl ("/bin/mount", "/bin/mount", point_de_montage, NULL);
```

et

```
execl ("/bin/umount", "/bin/umount", point_de_montage, NULL);
```

Cela simplifiera les vérifications des autorisations et rejettera sur l'application mount le problème de compatibilité des systèmes de fichiers avec la version du noyau. La configuration du logiciel – par le biais d'un fichier d'initialisation – ou d'une variable d'environnement peut permettre de préciser le chemin d'accès de mount et de umount pour le cas où ils ne se trouvent pas dans /bin (ce qui est peu probable car ils doivent être disponibles avec le minimum vital du système).

Journalisation

Linux incorpore plusieurs mécanismes de journalisation des informations. Tout d'abord, nous étudierons le mécanisme servant à connaître le nom des utilisateurs connectés au système. Ensuite, nous examinerons le fichier permettant de garder une trace de toutes les connexions et redémarrages. Enfin, nous pourrons voir l'utilisation du système *syslog*, qui permet d'afficher et de mémoriser tous les événements importants en provenance d'une application système.

Journal utmp

Le fichier **utmp** se trouve généralement dans le répertoire /var/run. On peut parfois le rencontrer dans /var/adm, voire dans /etc suivant les distributions. Il sert à mémoriser l'état présent du système en stockant des lignes horodatées contenant une description d'un certain nombre d'événements.

Le fichier utmp est mis à jour par tous les utilitaires système (init, getty, login, xterm, telnetd…). Chacun de ces processus renseignant au fur et à mesure la ligne de utmp qui le concerne.

Pour décrire ces données, la structure utmp est définie dans <utmp.h> :

Nom	Type	Signification
ut_type	short int	Description de l'enregistrement. Il existe dix types différents, que nous examinerons ci-dessous.
ut_pid	pid_t	PID du processus concerné.
ut_line	char [32]	Fichier spécial de terminal (sans le préfixe /dev).
ut_id	char [4]	Chaîne d'identification au sein du fichier /etc/inittab.
ut_user	char [32]	Nom de connexion de l'utilisateur.
ut_host	char [256]	Nom de l'hôte d'où provient la connexion.
ut_exit	struct exit_status	État de fin du processus.
ut_session	long int	Identifiant de session.
ut_tv	struct timeval	Horodatage de l'événement.
ut_addr_v6	int32_t [4]	Adresse IP de l'hôte distant.

Bien entendu, ces champs n'ont pas tous une signification simultanément. Leur utilisation dépend du type d'enregistrement.

Les différents types d'événements possibles sont les suivants :

Nom	Signification
EMPTY	Enregistrement vide.
RUN_LVL	Changement de niveau d'exécution du système.
BOOT_TIME	Démarrage de la machine. Permet d'enregistrer l'heure de boot.
OLD_TIME	Ancienne heure, juste avant une modification de l'horloge interne. Cet enregistrement est suivi d'un enregistrement NEW_TIME.
NEW_TIME	Nouvelle heure, juste après la modification de l'horloge interne.
INIT_PROCESS	Processus lancé par init.
LOGIN_PROCESS	Processus login.
USER_PROCESS	Connexion d'un utilisateur.
DEAD_PROCESS	Fin d'un processus.
ACCOUNTING	Début de comptabilisation.

Pour comprendre le principe, reprenons à la mise en route de la machine :

1. Un enregistrement BOOT_TIME est ajouté dès que le noyau a lancé le processus init. Celui-ci détermine alors son niveau d'exécution et ajoute un enregistrement RUN_LVL.
2. Le processus init consulte alors le fichier /etc/inittab et, pour chaque entrée valide, il invoque fork() et exec() pour lancer l'application attendue (par exemple getty ou xdm), en ajoutant à chaque fois un enregistrement INIT_PROCESS avec les champs ut_id et ut_pid remplis.
3. L'utilitaire getty recherche l'enregistrement correspondant à son PID, remplit son champ ut_line avec le nom du terminal qu'il surveille, et modifie le champ ut_type pour qu'il contienne LOGIN_PROCESS.
4. Lorsqu'un utilisateur se connecte, getty exécute login, qui recherche l'enregistrement correspondant à son PID, remplit le champ ut_user et modifie le champ ut_type avec le type USER_PROCESS.
5. L'utilitaire login peut aussi remplir les champs ut_host et ut_addr_v6 lorsqu'il a été invoqué par telnetd plutôt que par getty.
6. Lorsqu'un processus se termine, init en est informé grâce à l'appel-système wait(), il remplit l'enregistrement correspondant à son PID avec les codes de retour et lui met un type DEAD_PROCESS. Les applications comme getty ou xterm recherchent d'abord s'il existe un enregistrement de ce type correspondant à leur fichier spécial avant d'en créer un nouveau.

Il n'y a en effet pas de moyen d'effacer un enregistrement. Leur nombre est limité de fait par la quantité de pseudo-terminaux permettant des connexions simultanées.

Pour lire les informations de utmp, on utilise les fonctions **setutent()** et **endutent()**, qui initialisent et terminent la lecture, puis les routines **getutent()** ou sa contrepartie réentrante, **getutent_r()**, pour balayer le fichier séquentiellement.

```
void setutent (void);
void endutent (void);
struct utmp * getutent (void);
int getutent_r (struct utmp * utmp, struct utmp ** retour);
```

Lorsqu'on recherche un enregistrement correspondant à un PID donné, comme le fait login, on utilise **getutid()** ou **getutid_r()**, qui prennent en argument une structure utmp. Celle-ci doit avoir des champs ut_pid et ut_type remplis. Si le champ ut_type contient une constante xxx_PROCESS, on cherche un enregistrement ayant le même ut_pid et un ut_type correspondant également à un xxx_PROCESS. Sinon, on recherche un enregistrement ayant le même ut_type et le même ut_pid. Si aucune entrée ne correspond, getutid () renvoie NULL, et getutid_r() –1.

```
struct utmp * getutid (const struct utmp * utmp);
int getutid_r (struct utmp * utmp, struct utmp ** retour);
```

On peut aussi rechercher un enregistrement correspondant à un terminal particulier. Les fonctions **getutline()** et **getutline_r()** prennent une structure utmp en argument et renvoient l'enregistrement correspondant à la même valeur de ut_line, avec un ut_type contenant LOGIN_PROCESS ou USER_PROCESS.

```
struct utmp * getutline (const struct utmp * utmp);
int getutline_r (struct utmp * utmp, struct utmp ** retour);
```

Finalement, nous pouvons ajouter un enregistrement dans la base de données `utmp`. La fonction **`pututline()`** prend une structure `utmp` en argument et met à jour le fichier en recherchant un éventuel enregistrement avec le même PID. Le pointeur renvoyé est dirigé vers une copie de la structure ajoutée, ou est `NULL` si le processus n'a pas les autorisations nécessaires pour modifier la base de données `utmp`.

```
struct utmp * pututline (const struct utmp * utmp);
```

Le programme suivant affiche une partie des informations contenues dans la base de données `utmp`.

exemple_getutent.c :

```
    #include <stdio.h>
    #include <utmp.h>

    void
affiche_utmp (struct utmp * utmp)
{
    struct tm * tm;
    char        heure[80];

    tm = localtime(& (utmp->ut_tv.tv_sec));
    strftime(heure, 80, "%x %X", tm);
    switch (utmp->ut_type) {
        case EMPTY :
            break;
        case RUN_LVL :
            printf("%s : ", heure);
            printf("Run-level \n");
            break;
        case BOOT_TIME :
            printf("%s : ", heure);
            printf("Boot \n");
            break;
        case OLD_TIME :
            printf("%s : ", heure);
            printf("Old Time \n");
            break;
        case NEW_TIME :
            printf("%s : ", heure);
            printf("New Time \n");
            break;
        case INIT_PROCESS :
            printf("%s : ", heure);
            printf("Init process, ");
            printf("PID = %u, ", utmp->ut_pid);
            printf("inittab = %s\n", utmp->ut_id);
            break;
        case LOGIN_PROCESS :
            printf("%s : ", heure);
            printf("Login process, ");
            printf("PID = %u, ", utmp->ut_pid);
```

```
            printf("TTY = %s\n", utmp->ut_line);
            break;
        case USER_PROCESS :
            printf("%s : ", heure);
            printf("User process, ");
            printf("PID = %u, ", utmp->ut_pid);
            printf("TTY = %s, ", utmp->ut_line);
            printf("%s \n", utmp->ut_user);
            break;
        case DEAD_PROCESS :
            break;
        default :
            printf ("?");
            break;
    }
}

    int
main (void)
{
    struct utmp * utmp;
    while ((utmp = getutent()) != NULL)
        affiche_utmp(utmp);
    return EXIT_SUCCESS;
}
```

Voici un exemple sur une station Linux simple :

```
$ ./exemple_getutent
01/04/05 10:04:32 : Boot
01/04/05 10:04:32 : Run-level
01/04/05 10:04:52 : Login process, PID = 540, TTY = tty1
01/04/05 10:04:52 : Login process, PID = 541, TTY = tty2
01/04/05 10:04:52 : Login process, PID = 542, TTY = tty3
01/04/05 10:04:52 : Login process, PID = 543, TTY = tty4
01/04/05 10:04:52 : Login process, PID = 544, TTY = tty5
01/04/05 10:04:52 : Login process, PID = 545, TTY = tty6
01/04/05 10:04:52 : Init process, PID = 546, inittab = x
01/04/05 10:05:52 : User process, PID = 563, TTY = :0, ccb
01/06/05 15:09:49 : User process, PID = 1530, TTY = pts/1, ccb
$
```

Nous remarquons les processus `getty` qui tournent sur les six consoles virtuelles, ainsi que le démon `xdm` qu'on reconnaît grâce à l'identificateur *x* présent dans `/etc/inittab`. Enfin, deux connexions se font, l'une directement depuis un `xterm` (`TTY=:0`), et la seconde par un `telnet` depuis le réseau.

Fonctions X/Open

Une partie des routines que nous avons étudiées dispose d'équivalents portables car ils sont déclarés dans SUSv3. Ces fonctions manipulent les enregistrements `utmp` par l'intermédiaire d'une structure **`utmpx`** ayant les membres suivants (identiques à ceux de la structure `utmp`).

Nom	Type
ut_type	short int
ut_pid	pid_t
ut_line	char [32]
ut_id	char [4]
ut_user	char [32]
ut_tv	struct timeval

Les fonctions suivantes sont équivalentes à leurs homologues traitant les structures utmp. Elles sont d'ailleurs définies dans la bibliothèque GlibC par des alias. Notons qu'il n'existe pas de fonction réentrante :

```
void setutxent (void);
void endutxent (void);
struct utmpx * getutxent (void);
struct utmpx * getutxid (const struct utmpx * utmpx);
struct utmpx * getutxline (const struct utmpx * utmpx);
struct utmpx * pututxline (const struct utmpx * utmpx);
```

Toutes ces fonctions sont déclarées dans <utmpx.h>.

Journal wtmp

Le fichier utmp est effacé à chaque démarrage de la machine, et lorsqu'un processus de connexion se termine, son enregistrement est marqué comme DEAD_PROCESS avant d'être réutilisé ensuite. Il existe également sur le système Linux un fichier nommé **wtmp** qui n'est pas effacé. Les enregistrements y sont ajoutés successivement. Ce fichier sert d'historique des connexions. On le trouve en général dans /var/log, mais on peut aussi le rencontrer dans /var/adm ou /etc. Sur certaines machines, il est également copié automatiquement sur une imprimante système afin de conserver une trace de toutes les connexions des utilisateurs, à des fins de sécurité. Le fichier wtmp est souvent archivé ou effacé de façon automatique une fois par mois par un script d'administration déclenché par le démon crond.

Pour examiner le contenu de wtmp, on peut utiliser les mêmes routines que celles que nous avons déjà étudiées. Pour cela, il existe une routine nommée **utmpname()** qui permet de préciser le nom du fichier qu'on veut lire. Par défaut, c'est le fichier utmp du système qui est utilisé, mais cette fonction permet d'en indiquer un autre.

```
int utmpname (const char * fichier);
```

Pour savoir où se trouve le fichier à lire, on peut employer l'une des macros **_PATH_UTMP** et **_PATH_WTMP**, qui se transforment en chaînes de caractères représentant le chemin du fichier utmp ou wtmp.

Dans l'exemple suivant, nous fournissons les noms de fichiers sur la ligne de commande. Le programme emploie la routine affiche_utmp() de l'exemple précédent, que nous ne réécrivons pas ici.

exemple_utmpname.c :

```
    #include <stdio.h>
    #include <utmp.h>

    int
main (int argc, char * argv[])
{
    struct utmp * utmp;
    int           i;
    for (i = 1; i < argc; i ++) {
        if (utmpname(argv[1]) != 0)
            continue;
        while ((utmp = getutent()) != NULL)
            affiche_utmp(utmp);
    }
    return EXIT_SUCCESS;
}
```

Ce programme offre le même genre de résultat que l'utilitaire `last`.

```
$ ./exemple_utmpname /var/log/wtmp
01/04/05 20:47:43 : Boot
01/04/05 20:47:43 : Run-level
01/04/05 20:47:43 : Init process, PID = 127, inittab = l5
01/04/05 20:48:03 : Init process, PID = 526, inittab = ud
01/04/05 20:48:03 : Init process, PID = 527, inittab = 1
01/04/05 20:48:03 : Init process, PID = 528, inittab = 2
01/04/05 20:48:03 : Init process, PID = 529, inittab = 3
...
01/06/05 10:05:52 : User process, PID = 563, TTY = :0, ccb
01/06/05 10:58:42 : User process, PID = 795, TTY = ftpd795, ccb
01/06/05 11:00:09 : User process, PID = 806, TTY = pts/1, ccb
01/06/05 15:09:49 : User process, PID = 1530, TTY = pts/1, ccb
01/06/05 15:36:52 : User process, PID = 1580, TTY = ftpd1580, ccb
01/06/05 16:12:20 : User process, PID = 1631, TTY = ftpd1631, ccb
$
```

Notre routine d'affichage des enregistrements n'est peut-être pas très adaptée à la présentation de traces de connexions, car nous devrions plutôt éliminer les lignes contenant les enregistrements `INIT_PROCESS` ou `LOGIN_PROCESS` qui correspondent aux `getty` et `login`, et conserver les `DEAD_PROCESS` qui représentent la déconnexion d'un utilisateur.

Pour ajouter un enregistrement au fichier `wtmp`, on n'utilisera pas `pututline()`, car elle écrase les entrées inutilisées, mais plutôt **`updwtmp()`**, qui écrit simplement la nouvelle ligne de données à la fin du fichier.

```
void updwtmp (const char * fichier, const struct utmp * utmp);
```

Journal syslog

Une application, même si elle fonctionne en arrière-plan, doit pouvoir communiquer des informations de temps à autre. L'écriture sur `stdout` ou `stderr` n'est pas toujours possible, notamment pour les logiciels fonctionnant sous forme de démons. Pour pouvoir transmettre

des indications sur son état, un programme peut alors employer plusieurs techniques, comme l'émission d'un courrier électronique à destination de l'utilisateur qui l'a lancé, ou proposer une connexion réseau (par `telnet`) et afficher ainsi sa configuration.

Une autre méthode, plus souple, consiste à utiliser le démon `syslogd`. Celui-ci est lancé au démarrage par les scripts d'initialisation du système et il reste en attente de messages. Les fonctions de bibliothèque **`openlog()`**, **`syslog()`** et **`closelog()`** permettent de lui transmettre des données. En fonction de la configuration du démon (*via* `/etc/syslog.conf`) et de la gravité du message, celui-ci peut être stocké dans un fichier, envoyé dans un tube nommé vers un autre programme (en général `mail`), affiché sur la console et sur les écrans des utilisateurs, ou même transmis par le réseau à destination d'un autre démon `syslogd` fonctionnant sur une machine de supervision.

Ces trois routines sont disponibles dans `<syslog.h>` :

```
void openlog  (char * identificateur, int option, int type);
void syslog   (int urgence, char * format, ...);
void closelog (void);
```

La fonction `openlog()` permet d'ouvrir une session de journalisation. Le premier argument est un identificateur qui sera ajouté à chaque message pour le distinguer. En général, on choisit le nom du programme.

Le second argument peut contenir une ou plusieurs des constantes symboliques suivantes, liées par un OU binaire :

Nom	Signification
`LOG_CONS`	Écrire les messages sur la console système si une erreur se produit lors de leur traitement.
`LOG_NDELAY`	Ouvrir tout de suite la communication avec le démon `syslogd`, sans attendre l'arrivée du premier message.
`LOG_PERROR`	Envoyer sur la sortie d'erreur une copie des messages.
`LOG_PID`	Ajouter le PID du processus appelant dans chaque message.

L'utilisation de `LOG_PERROR` permet de simplifier la mise au point d'un programme qu'on fera fonctionner ensuite sous forme de démon. De même, `LOG_PID` est très utile car cette information est souvent indispensable, et on évite ainsi de devoir l'inscrire explicitement dans chaque message.

Enfin, le troisième argument de `openlog()` est une valeur numérique servant à classer le programme dans une catégorie de logiciels. Cela permet de filtrer les messages, par exemple en redirigeant tous ceux qui concernent le courrier vers l'utilisateur `postmaster`. Les constantes suivantes sont déclarées dans `<syslog.h>` :

Nom	Utilisation
`LOG_KERN`	Message provenant du noyau
`LOG_USER`	Message provenant d'une application utilisateur
`LOG_MAIL`	Système de gestion du courrier électronique
`LOG_DAEMON`	Ensemble des démons du système

Nom	Utilisation
LOG_AUTH	Système d'authentification des utilisateurs
LOG_SYSLOG	Démon syslogd lui-même
LOG_LPR	Système de gestion des impressions
LOG_NEWS	Système des news Usenet
LOG_UUCP	Message provenant d'un démon uucp
LOG_CRON	Exécution différée par crond
LOG_AUTHPRIV	Système d'authentification personnel
LOG_FTP	Démon ftpd
LOG_LOCAL0 …LOG_LOCAL7	Message provenant d'une application spécifique du système

On emploiera généralement LOG_USER ou LOG_LOCAL0 à LOG_LOCAL7 pour les logiciels personnels.

En fait, openlog() ne fait qu'initialiser des champs qui seront utilisés ensuite lors de la transmission effective des messages. Pour en envoyer un, on emploie syslog(). Cette fonction prend un premier argument qui correspond à l'urgence du message. On définit les niveaux de priorité suivants :

Nom	Signification
LOG_EMERG	Le système concerné n'est plus utilisable.
LOG_ALERT	L'intervention immédiate d'un administrateur est indispensable ou le système va devenir inutilisable.
LOG_CRIT	Des conditions critiques se présentent, pouvant nécessiter une intervention.
LOG_ERR	Des erreurs ont été détectées.
LOG_WARNING	Des conditions rares ou inattendues ont été observées.
LOG_NOTICE	Information importante, mais fonctionnement normal.
LOG_INFO	Information sans importance renseignant sur l'état du système.
LOG_DEBUG	Données utiles pour le débogage, à ignorer sinon.

On notera que le terme « système » dans ce tableau fait référence à l'application invoquant syslog() et éventuellement à ses interlocuteurs, mais qu'il ne regroupe pas l'ensemble des fonctionnalités de la machine comme on l'entend habituellement.

Le format se trouvant en second argument de syslog() ainsi que les autres arguments éventuels correspondent exactement à ceux de printf(). La seule différence est l'existence d'un code %m qui est remplacé par la chaîne strerror(errno).

Si openlog() n'a pas encore été appelée, syslog() l'invoque automatiquement avec les arguments :

- identificateur = NULL ;
- option = 0 ;
- type = LOG_USER.

La routine `closelog()`ferme la session de communication avec `syslogd`. Cette fonction n'est pas indispensable, la session étant terminée automatiquement à la fin du programme.

Pour des raisons de sécurité – comme nous l'avons déjà mentionné avec `printf()` – il faut toujours utiliser `syslog(urgence, "%s", message)` et jamais `syslog(urgence, message)` si le message est susceptible d'être fourni par l'utilisateur. Ceci concerne surtout les démons et les applications réseau.

Voici un exemple simple utilisant `syslog()`.

exemple_syslog.c :

```
    #include <stdio.h>
    #include <syslog.h>

    int
main (int argc, char * argv[])
{
    int i;

    openlog(argv[0], LOG_PID, LOG_USER);
    for (i = 1; i < argc; i ++)
        syslog(LOG_INFO, argv[i]);
    closelog();

    return EXIT_SUCCESS;
}
```

À l'exécution de ce programme les messages sont redirigés, sur notre système, vers le fichier `/var/log/messages`, lisible uniquement par *root*.

```
$ ./exemple_syslog "premier message"
$ ./exemple_syslog "deuxième message"
$ su
Password:
# tail /var/log/messages
   [...]
Jan 06 17:37:12 venux ./exemple_syslog[1807]: premier message
Jan 06 17:37:19 venux ./exemple_syslog[1808]: deuxième message
   [...]
#
```

L'utilisation de `syslog()` est fortement recommandée pour le développement d'applications fonctionnant essentiellement en arrière-plan, car cette routine laisse à l'administrateur du système le choix du comportement vis-à-vis des messages de diagnostic et d'erreur. Cette souplesse est très appréciable car on peut ainsi plus facilement décider d'éliminer tous les messages peu importants, de les stocker dans un fichier ou de les rediriger vers une console réservée à cet usage.

Conclusion

Nous avons vu dans ce chapitre les méthodes pour accéder à de nombreuses informations concernant le système. Une large partie d'entre elles sont suffisamment portables pour être disponibles sur l'essentiel des systèmes Unix.

Les fonctions présentées ici sont certainement suffisantes pour réaliser la plus grande partie des tâches d'administration du système. On trouvera plus de détails sur ces travaux ainsi que sur les différents outils d'aide à l'administrateur dans [FRISCH 2003] *Les bases de l'administration système*.

L'utilisation des fonctionnalités du démon `syslogd` dans un programme lui apporte une souplesse notable. Pour la configuration du démon proprement dit, on pourra se reporter aux pages de manuel `syslogd(8)` et `syslog.conf(5)`.

27

Internationalisation

Nous avons déjà signalé dans le chapitre 23, à propos des caractères larges, que les développeurs se soucient de plus en plus des possibilités d'internationalisation de leurs logiciels. En dehors des applications « maison », destinées à un usage unique et très spécifique, la plupart des programmes peuvent voir subitement leur portée étendue à une échelle internationale grâce à Internet par exemple.

En raison du volume de discussion qu'engendrent les problèmes d'internationalisation, un sigle a même été créé, *i18n*, signifiant « *i* suivi de 18 lettres puis d'un *n* », afin d'éviter les guerres de clans entre les partisans du mot *internationalisation* et ceux – américains – du terme *internationalization*.

Après avoir présenté les principes de l'internationalisation, nous examinerons des méthodes permettant d'offrir des messages d'interface dans la langue de l'utilisateur. Malgré tout, l'internationalisation d'un logiciel ne consiste pas uniquement en la traduction des messages de l'interface utilisateur, même s'il s'agit probablement du point le plus important dans la plupart des cas. En fait, la langue n'est qu'une partie des conventions culturelles propres à un peuple, et l'ordre de présentation des éléments d'une date est par exemple un autre aspect de l'internationalisation d'une application.

Pour permettre la transposition d'un système vers d'autres pays, la bibliothèque C autorise l'utilisateur à configurer ces éléments culturels et linguistiques à son gré. L'adaptation aux désirs de l'utilisateur se fait par le biais de la *localisation*[1].

Nous verrons dans ce chapitre comment employer l'ensemble des éléments configurés dans la localisation.

1. Le mot anglais *locale* est traduit différemment suivant les auteurs. Je conserverai le terme *localisation*, qui est le plus répandu même s'il n'est pas très esthétique.

Principe

La localisation est un ensemble de règles, réparties par catégories, que la bibliothèque C applique dans les routines qui doivent réagir différemment suivant les choix de l'utilisateur. Par exemple, il existe une catégorie dans laquelle on indique le caractère qu'on préfère utiliser pour séparer la partie entière d'un nombre réel de ses décimales. Dans la localisation anglo-saxonne, il s'agit du point, alors que dans la localisation française on préfère la virgule. Certaines routines d'affichage comme `printf()` prennent cette information en considération pour présenter leurs résultats.

La plupart des utilitaires du système sont sensibles à la localisation, du moins en ce qui concerne la traduction des messages. Nous pourrons donc observer directement quelques effets des modifications apportées. Pour configurer sa localisation, un utilisateur remplit des variables d'environnement qui seront consultées par les applications lancées par la suite.

Comme nous l'avons précisé dans le chapitre 3, les variables d'environnement ne concernent que le processus qui les configure et ses descendants. Si l'utilisateur définit sa localisation dans une session shell, toutes les applications lancées ensuite grâce à ce shell en bénéficieront, mais pas les logiciels démarrés depuis un autre shell ou depuis un environnement graphique X-Window. L'administrateur du système configure souvent une localisation par défaut dans les fichiers d'initialisation communs à tous les utilisateurs. Il s'agit généralement de la localisation correspondant à l'implantation physique de la station. Chaque utilisateur peut toutefois modifier cette configuration dans ses propres scripts de connexion afin de l'adapter à ses préférences.

Pour bénéficier automatiquement de l'internationalisation des routines de la bibliothèque C, il suffit d'insérer deux lignes dans une application :

- Le fichier d'en-tête `<locale.h>` doit être inclus en début de module.
- L'instruction `setlocale(LC_ALL, "")` doit être appelée en début de programme.

Rien qu'avec ces deux lignes un programme est capable de s'adapter correctement à la plupart des conventions culturelles de l'utilisateur, hormis la langue bien entendu. Pour obtenir une internationalisation au niveau du langage, il faut stocker les messages et leurs traductions dans des catalogues, comme nous le verrons plus loin.

Catégories de localisations disponibles

Nous avons indiqué que la localisation se faisait par l'intermédiaire de plusieurs catégories différentes. Ceci permet à l'utilisateur de configurer indépendamment plusieurs aspects de l'interface de l'application. Par exemple, il est possible de demander que les messages soient affichés en français pour faciliter la lecture, mais que les dates et les valeurs numériques soient affichées en respectant les normes américaines, afin de récupérer directement ces données pour les transmettre à des collègues étrangers.

Chaque catégorie est représentée par une variable d'environnement et par une constante symbolique du même nom, disponible au sein de l'application. Les catégories sont les suivantes :

Nom	Signification
LC_ALL	Cette catégorie écrase toutes les autres. On l'utilise pour donner une valeur immédiate de localisation à toutes les catégories. En réalité, l'emploi de cette variable d'environnement comme configuration est un peu abusif, elle ne devrait être utilisée qu'en tant que constante symbolique pour consulter la localisation.
LC_COLLATE	Dans cette catégorie se trouvent les règles employées par les routines devant ordonner des chaînes de caractères, comme `strcoll()`, que nous avons étudiée dans le chapitre 15.
LC_CTYPE	Cette catégorie concerne les routines de classification des caractères comme `isalpha()`, ainsi que celles de conversion comme `tolower()`. Elle sert également à déterminer les règles employées pour les conversions entre caractères larges et séquences multioctets, comme nous l'avons indiqué dans le chapitre 23.
LC_MESSAGES	La traduction des messages réclamée par la catégorie `LC_MESSAGES` concerne l'interface avec l'utilisateur. Il ne s'agit pas nécessairement de la même langue que celle qui est employée dans les données elles-mêmes ni surtout du même jeu de caractères.
LC_MONETARY	Cette catégorie configure la manière de représenter des valeurs monétaires, tant du point de vue du symbole évoquant la monnaie que pour la position de ce symbole, et la séparation entre partie entière et décimale.
LC_NUMERIC	Avec cette catégorie, on indique les coutumes de représentation des valeurs numériques, comme la séparation des chiffres par milliers ou le symbole utilisé comme séparateur décimal.
LANG	La variable `LANG` sert à définir la langue utilisée pour l'ensemble des messages et des textes, mais c'est surtout une valeur par défaut, qui permet de configurer toutes les catégories qui ne sont pas remplies explicitement.

La localisation est donc l'ensemble de toutes ces catégories, représentant chacune des règles usuelles appliquées à l'emplacement où se trouve l'utilisateur. Pour remplir une catégorie, on emploie une chaîne de caractères indiquant en premier lieu la langue choisie. Il s'agit de deux caractères minuscules, dont voici quelques exemples :

Nom	Langue
da	danois
de	allemand
el	grec
en	anglais
es	espagnol
fi	finnois
fr	français
it	italien
nl	hollandais
pt	portugais
sv	suédois

Le nom de la localisation est ensuite précisé par un emplacement géographique si plusieurs pays emploient la même langue, mais avec des différences de coutumes dans d'autres catégories.

Par exemple, si on demande l'affichage de la valeur monétaire 2000 avec une localisation francophone de France (`fr_FR`), on obtient

```
2 000,00€
```

alors que l'affichage avec la localisation francophone du Canada (`fr_CA`) donne

```
2 000,00$
```

La même valeur en anglais du Canada (`en_CA`) est affichée :

```
$2,000.00
```

En anglais de Grande-Bretagne (`en_GB`), le résultat est :

```
£2,000.00
```

Alors qu'en anglais commun au Royaume-Uni (`en_UK`), on voit :

```
2000.00
```

L'emplacement géographique est donc précisé avec un code de deux lettres majuscules. Voici quelques pays européens :

Pays	Code
Allemagne	DE
Angleterre	GB
Autriche	AT
Belgique	BE
Danemark	DK
Grèce	EL
Espagne	ES
Finlande	FI
France	FR
Irlande	IR
Italie	IT
Luxembourg	LU
Hollande	NL
Portugal	PT
Suède	SV

Une source fréquente d'erreur lors de la configuration de la localisation est l'inversion entre le pays et la langue[1]. Ainsi la disposition des majuscules dans "FR_fr" paraît plus naturelle que dans "fr_FR", mais c'est pourtant ce second cas seulement qui fonctionne, le premier

1. Disons la deuxième source d'erreur la plus fréquente, la première étant l'oubli pur et simple de l'appel à `setlocale (LC_ALL, "")` en début de programme…

étant inconnu (donc ignoré et équivalent à la localisation américaine par défaut). Dans certaines situations, on peut ajouter le nom d'un jeu de caractères à la suite, mais c'est plutôt rare.

> **Attention**
> Les noms des localisations proprement dites peuvent varier suivant les systèmes d'exploitation. Nous décrivons ici leur aspect avec la GlibC, mais il ne faut pas faire de suppositions hâtives quant au contenu des variables sur d'autres systèmes.

Voyons donc les effets de la localisation sur quelques utilitaires, en commençant par les messages d'erreur de `/bin/ls` :

```
$ unset LANG
$ unset LC_ALL
$ export LANG=fr_FR
$ ls inexistant
ls: inexistant: Aucun fichier ou répertoire de ce type
$ export LANG=en
$ ls inexistant
ls: inexistant: No such file or directory
$ export LC_MESSAGES=fr
$ ls inexistant
ls: inexistant: Aucun fichier ou répertoire de ce type
$
```

Nous allons observer à présent les répercussions de la localisation sur l'affichage de la date. On vérifiera que la catégorie `LC_ALL` a bien préséance sur `LANG`. Le format `"%x"` ordonne à l'utilitaire `/bin/date` d'afficher la représentation locale de la date. Le test a lieu le 8 mars.

```
$ unset LANG LC_ALL
$ export LANG=en
$ date +"%x"
03/08/00
$ export LC_ALL=fr_FR
$ date +"%x"
08.03.2000
$
```

Pour connaître les règles applicables dans une localisation donnée, la bibliothèque C dispose de fichiers de configuration placés en général dans les répertoires `/usr/locale/` ou `/usr/share/locale/`. On y trouve normalement un nombre important de sous-répertoires, chacun représentant une localisation connue par le système. Tout cela peut varier légèrement en fonction de la distribution Linux choisie. Les fichiers employés par la bibliothèque C sont dans un format binaire. Pour modifier une localisation existante – ou en créer une nouvelle –, il faut installer les sources des localisations. Elles sont généralement établies avec l'ensemble des sources de la GlibC. On peut invoquer `/usr/bin/localedef --help` pour savoir où les sources des localisations sont placées (par exemple `/usr/share/i18n/locale/`). Ce répertoire regroupe un ensemble de fichiers décrivant toutes les localisations connues par la bibliothèque. Ces fichiers sont tout à fait lisibles, leur format est assez intuitif. L'utilitaire `/usr/bin/localedef` sert à compiler l'un de ces fichiers en créant les répertoires système et les fichiers binaires nécessaires pour que la nouvelle localisation soit reconnue par la bibliothèque C. Cette tâche est normalement réservée à l'administrateur du système.

Traduction de messages

Le fait de proposer une interface dans la langue de l'utilisateur est probablement le premier souhait en ce qui concerne l'internationalisation d'un programme. La traduction automatique des messages d'une langue à l'autre n'est pas encore possible, aussi un programme doit-il employer pour ses messages un stock de libellés, et afficher ceux qui correspondent à la langue de l'utilisateur.

L'ensemble de tous les messages et leurs traductions peuvent être directement insérés dans le code source de l'application. La sélection du libellé correspondant à la traduction d'un message dans la langue désirée se fera en fonction d'un paramétrage interne (choix dans un menu) ou externe (variable d'environnement). Cette méthode est parfois employée lorsqu'un logiciel doit être distribué sous forme binaire sur des systèmes d'exploitation totalement différents, n'offrant pas toujours des possibilités d'internationalisation. Si cette approche se justifie donc dans certains cas, elle est quand même fortement déconseillée, car l'ajout du support d'une nouvelle langue ou la correction d'une faute de traduction d'un message nécessitent la recompilation de l'application.

On préfère donc regrouper les libellés dans un fichier externe, qu'on peut échanger au gré de la localisation.

Il y a un avantage supplémentaire au regroupement de tous les messages dans un unique fichier, même sans tenir compte des possibilités de traduction. Cela permet en effet d'avoir sous les yeux tous les libellés d'interface du logiciel et de s'assurer immédiatement de l'homogénéité de l'ensemble. Lorsqu'il y a plusieurs possibilités pour nommer un objet manipulé par le programme, ou s'il faut choisir entre traduire un nom ou laisser le terme original qu'on pourra retrouver dans d'autres logiciels, on s'assure, en voyant tous les messages côte à côte, que les mêmes décisions ont été prises pour toute l'interface.

La bibliothèque GlibC offre deux méthodes différentes pour gérer un ensemble de messages externes, stockés dans des fichiers qui pourront être mis à jour sans que l'application ait besoin d'être recompilée. Ces deux dernières nécessitent d'abord que la bibliothèque puisse trouver le fichier de messages lui appartenant, adapté à la langue choisie. Les différences apparaissent ensuite dans la manière d'accéder aux libellés contenus dans le fichier proprement dit.

Catalogues de messages gérés par catgets()

Ce premier mécanisme est plus ancien et plus répandu que le second. Il rend aussi le travail du développeur sensiblement plus compliqué. Les fonctions `catopen()`, `catgets()` et `catclose()` que nous allons examiner sont définies dans les spécifications SUSv3. Chaque message du logiciel doit être associé à une clé numérique unique. Ceci représente le point le plus complexe, principalement lorsque le développement d'une application se fait de manière répartie avec plusieurs équipes indépendantes.

Tout d'abord, le programme doit ouvrir le catalogue de messages. Ceci s'effectue avec la fonction **`catopen()`**, déclarée ainsi dans `<nl_types.h>` :

```
nl_catd catopen (const char * nom, int attribut);
```

Cette routine essaye d'ouvrir le catalogue dont le nom est passé en premier argument. Si ce nom contient un caractère « / », on considère qu'il s'agit d'un chemin d'accès entier. Sinon,

on suppose qu'il s'agit du nom d'un catalogue qui est alors recherché dans divers répertoires suivant la configuration des variables d'environnement. Le fait d'employer un chemin figé ne se justifie que lors de la mise au point du programme, car le principe même de l'internationalisation consiste à laisser l'utilisateur configurer le répertoire correspondant à sa localisation.

La recherche se fait en employant la variable d'environnement `NLSPATH`. Celle-ci contient un ou plusieurs chemins d'accès séparés par des deux-points. Un chemin peut comprendre des codes spéciaux, qui seront remplacés automatiquement lors de la tentative d'ouverture :

Code	Signification
`%N`	Nom du catalogue tel qu'il a été transmis en premier argument de `catopen()`.
`%L`	Localisation configurée pour les messages d'interface.
`%l`	Langage configuré pour les messages d'interface, sans préciser l'emplacement ni le jeu de caractères. Par exemple `fr` dans la localisation `fr_BE.ISO-8859-1`.
`%t`	Emplacement configuré pour les messages d'interface, sans préciser la langue ni le jeu de caractères. `BE` dans la localisation `fr_BE.ISO-8859-1`.
`%c`	Jeu de caractères configuré pour les messages d'interface, sans préciser la langue ni l'emplacement géographique. `ISO-8859-1` dans la localisation `fr_BE.ISO-8859-1`.
`%%`	Le caractère `%` lui-même.

L'attribut indiqué en seconde position lors de l'appel de `catopen()` permet de préciser les variables prises en compte pour la localisation. Si cet attribut est nul, la fonction n'emploie que la variable `LANG`. Sinon, si l'attribut prend la valeur **`NL_CAT_LOCALE`**, la bibliothèque recherche la localisation successivement dans les variables `LC_ALL`, `LC_MESSAGES` et `LANG`. On emploiera donc toujours `NL_CAT_LOCALE` en second argument de `catopen()`.

Si la variable `NLSPATH` n'est pas définie, la fonction emploie une valeur par défaut, configurée lors de la compilation de la bibliothèque C, qui correspond généralement à :

```
/usr/share/locale/%L/%N:/usr/share/locale/%L/LC_MESSAGES/%N:/usr/share/locale/%l
➥ /%N:/usr/share/locale/%l/LC_MESSAGES/%N
```

Cela signifie que lors d'une tentative d'ouverture du fichier de catalogue `msg`, dans la localisation `fr_FR`, le système recherche le fichier successivement dans :

```
/usr/share/locale/fr_FR/msg
/usr/share/locale/fr_FR/LC_MESSAGES/msg
/usr/share/locale/fr /msg
/usr/share/locale/fr/LC_MESSAGES/msg
```

Le descripteur de catalogue est du type opaque `nl_catd` et sera employé dans les fonctions `catgets()` et `catclose()`. Si aucun fichier n'est disponible, `catopen()` renvoie `(nl_catd) -1`.

Une fois que le catalogue de messages est ouvert, on peut accéder à son contenu à l'aide de la fonction **`catgets()`**, déclarée ainsi :

```
char * catgets (nl_catd catalogue,
                int ensemble, int message,
                const char * original);
```

Cette fonction recherche dans le catalogue décrit par son premier argument – obligatoirement obtenu grâce à `catopen()` – le message appartenant à l'ensemble indiqué en deuxième position, et dont le numéro est passé en troisième argument. Si le message n'est pas disponible, la chaîne transmise en dernier argument est renvoyée.

Cette chaîne est donc rédigée dans une langue par défaut, la plupart du temps en anglais. L'organisation du catalogue sous forme d'ensembles de messages permet de découper l'application en ensembles fonctionnels, en attribuant un numéro général à chaque équipe de développement afin qu'elle maintienne elle-même la numérotation dans son propre ensemble. Les numéros peuvent être choisis arbitrairement, ils n'ont pas besoin de se suivre. Par contre, la paire (ensemble, numéro) ne peut désigner qu'un seul message dans le catalogue.

Ceci complique sensiblement le travail des programmeurs, qui doivent gérer une nomenclature supplémentaire dans leurs applications.

Pour refermer un catalogue de messages qui n'est plus utilisé, on emploie **`catclose()`**, déclarée ainsi :

```
int catclose (nl_catd catalogue);
```

Cette fonction renvoie 0 si tout s'est bien passé, et –1 si une erreur s'est produite (généralement c'est le descripteur de catalogue qui est erroné).

Les catalogues sont créés à l'aide de l'utilitaire **`/usr/bin/gencat`**. Celui-ci prend en entrée un fichier de texte contenant les chaînes de caractères et fabrique un fichier binaire permettant l'accès rapide avec `catgets()`. Le détail de ce fichier binaire ne concerne que la bibliothèque C. Le format des fichiers lus par `gencat` est décrit en détail dans la documentation Gnu. Voyons-en les principales caractéristiques :

- Les lignes blanches ou commençant par un symbole `$` suivi d'un caractère blanc sont ignorées. On peut introduire ainsi des commentaires.
- Une ligne ayant la forme `$set <identifiant>` indique le début d'un ensemble de messages.
- Une ligne ayant la forme `<identifiant> <chaîne de caractères>` précise un message appartenant à l'ensemble en cours.

L'identifiant de l'ensemble ou celui du message peuvent être signalés sous forme numérique, mais également sous forme de mot-clé alphanumérique. Dans ce cas, `gencat` les remplacera par des numéros adéquats, et créera un fichier où les identifiants seront définis sous forme de constantes symboliques. On pourra donc inclure ce fichier en début de programme. Pour créer les constantes symboliques, `gencat` ajoute le suffixe `Set` à la fin des noms d'ensembles et insère le nom de l'ensemble devant les chaînes. Le programme suivant va permettre de mieux comprendre ce principe.

exemple_catgets.c :

```
    #include <nl_types.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include "exemple_catgets.h"

    int
main (void)
{
    nl_catd catalogue;
```

```
    char * chaine;

    if ((catalogue = catopen("msg_catgets", NL_CAT_LOCALE))
              == (nl_catd) -1) {
        fprintf(stderr, "unable to open catalog \n");
        exit(EXIT_FAILURE);
    }
    chaine = catgets(catalogue, premier_Set, premier_chaine_1,
                "This is the first string in the first set");
    fprintf(stdout, "%s \n", chaine);
    chaine = catgets(catalogue, premier_Set, premier_chaine_2,
                "and here is the second string in the first set.");
    fprintf(stdout, "%s \n", chaine);
    chaine = catgets(catalogue, second_Set, second_chaine_1,
                "Now let's have a look at the 1st string in 2nd set,");
    fprintf(stdout, "%s \n", chaine);
    chaine = catgets(catalogue, second_Set, second_chaine_2,
            "and finaly the second string in the second set.");
    fprintf(stdout, "%s \n", chaine);
    catclose(catalogue);
    return EXIT_SUCCESS;
}
```

Nous construisons aussi le fichier de messages traduits en français ainsi :

exemple_catgets.msg :

```
$    Voici le catalogue de messages pour
$    l'exemple_catgets.

$set premier_
chaine_1 Ceci est la première chaîne du premier ensemble,
chaine_2 et voici la seconde chaîne du premier ensemble.

$     Nous pouvons insérer des commentaires
$     qui seront ignorés

$set second_
chaine_1 Maintenant voyons la 1ere chaîne du 2eme ensemble,
chaine_2 et finalement la seconde chaine du second ensemble.
```

À présent, nous compilons le catalogue de messages, en demandant à `gencat` de nous fournir aussi un fichier de définition des constantes :

```
$ gencat -o msg_catgets -H exemple_catgets.h exemple_catgets.msg
$ cat exemple_catgets.h
#define second_Set 0x2  /* exemple_catgets.msg:11 */
#define second_chaine_1 0x1     /* exemple_catgets.msg:12 */
#define second_chaine_2 0x2     /* exemple_catgets.msg:13 */

#define premier_Set 0x1 /* exemple_catgets.msg:4 */
#define premier_chaine_1 0x1    /* exemple_catgets.msg:5 */
#define premier_chaine_2 0x2    /* exemple_catgets.msg:6 */
$
```

Nous pouvons maintenant compiler l'application et installer le fichier `msg_catgets` dans le répertoire `/usr/share/locale/fr/LC_MESSAGES/` afin qu'il soit trouvé par la bibliothèque C dans la localisation correspondante.

```
$ ls msg_catgets
msg_catgets
$ su
Password:
# cp msg_catgets /usr/share/locale/fr/LC_MESSAGES/
# exit
exit
$ unset LC_ALL LC_MESSAGES LANG
$ ./exemple_catgets
This is the first string in the first set
and here is the second string in the first set.
Now let's have a look at the 1st string in 2nd set,
and finaly the second string in the second set.
$ export LANG=fr_FR
$ ./exemple_catgets
Ceci est la première chaîne du premier ensemble,
et voici la seconde chaîne du premier ensemble.
Maintenant voyons la 1ere chaîne du 2eme ensemble,
et finalement la seconde chaine du second ensemble.
$
```

Nous voyons que ce mécanisme fonctionne très bien mais qu'il est très lourd à mettre en œuvre dans le codage du programme, chaque manipulation de chaîne devant faire l'objet de vérifications dans la nomenclature pour connaître le nom ou le numéro de l'ensemble et celui du message.

Il existe pourtant une alternative plus simple : les fonctionnalités *GetText* du projet Gnu.

Catalogues de messages Gnu GetText

Le principe des catalogues de messages *GetText* est d'employer la chaîne originale comme clé d'accès dans le catalogue de traduction. Ainsi, il n'y a plus besoin de manipuler des identificateurs, puisque la chaîne se suffit à elle-même.

Le projet *Gnu GetText* est relativement ambitieux puisqu'il contient de nombreux outils pour aider à internationaliser des programmes qui n'étaient pas conçus pour l'être à l'origine. Nous allons simplement présenter ici les fonctionnalités qui concernent le programmeur désireux d'employer *GetText* comme une alternative plus pratique à `catgets()`.

L'ensemble de la traduction repose essentiellement sur l'emploi d'une unique fonction, nommée **`gettext()`**, et déclarée dans `<libintl.h>` :

```
char * gettext (const char * origine);
```

L'interface de cette routine se rapproche au maximum de celle que pourrait proposer – que proposera peut-être un jour – un traducteur automatique. On lui transmet la chaîne originale et elle renvoie un pointeur sur une zone de mémoire statique contenant la traduction adaptée à la localisation de l'utilisateur. Si la traduction est impossible ou si la localisation est la même que celle du concepteur du programme, le pointeur renvoyé est identique à celui de la chaîne transmise.

On peut donc écrire des choses comme :

```
fprintf(stdout, gettext("Vitesse : %d bits / sec \n"),  vitesse);

fprintf(stdout, gettext("Parité = %s \n"),
       parite == PARITE_PAIRE ? gettext("paire") : gettext("impaire"));
```

On peut traduire aussi bien des chaînes de caractères correspondant à des messages que des formats pour `printf()`par exemple.

On devine que l'éventail des possibilités offertes par une telle interface est assez large. En effet, dans l'implémentation actuelle, la traduction est simplement recherchée dans un fichier, mais il est possible d'imaginer que la fonction `gettext()` peut évoluer pour interroger – par réseau – une énorme base de données ou un logiciel de traduction automatique. Dans le cas d'un portage sur un système ne supportant pas ce mécanisme, on définit simplement une macro

```
#define gettext(X)    (X)
```

en tête de programme pour annuler la tentative de traduction.

Pour que la bibliothèque puisse faire correspondre une traduction à un message, il faut lui indiquer le catalogue de messages à employer. Ceci s'effectue à l'aide de deux fonctions, `textdomain()` et `bindtextdomain()`. La bibliothèque *GetText* introduit en effet le concept de domaine, qui permet de scinder la base de textes en plusieurs fichiers. En général, une application n'utilise qu'un seul domaine, qu'elle configure dès le démarrage du programme. Ceci s'effectue avec **`textdomain()`** :

```
char * textdomain (const char * domaine);
```

Cette fonction signale à la bibliothèque que les messages ultérieurs seront recherchés dans le domaine dont on passe le nom. Ce nom sera utilisé pour déterminer le fichier contenant les libellés des messages. La fonction **`bindtextdomain()`** permet d'indiquer le nom du répertoire dans lequel se trouve l'arborescence des fichiers correspondant à un domaine particulier :

```
char * bindtextdomain (const char * domaine, const char * repertoire);
```

En fait, le fichier de traduction est recherché avec le chemin d'accès composé ainsi :

```
/répertoire_de_bindtextdomain()/localisation/LC_MESSAGES/domaine.mo
```

Le répertoire de départ a été spécifié avec `bindtextdomain()`. Il s'agit en général de `/usr/share/locale`. La localisation est extraite successivement des variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES` et `LANG`. Le nom du fichier final est construit avec le nom de domaine et le suffixe `.mo` signifiant *Machine Object*. Ce fichier est binaire, compilé par l'utilitaire `/usr/bin/msgfmt` à partir du fichier de texte avec le suffixe `.po` signifiant *Portable Object*.

Le projet *GetText* incorpore des macros pour permettre l'édition facile du fichier `.po`, mais son format est tellement simple que nous pourrons le manipuler directement.

Nous allons utiliser le même principe qu'avec `exemple_catgets`, en prenant cette fois-ci les fonctionnalités *GetText*.

exemple_gettext.c :

```
    #include <libintl.h>
    #include <stdio.h>
    #include <stdlib.h>
```

```
    int
main (void)
{
    textdomain("exemple_gettext");
    bindtextdomain("exemple_gettext", "/usr/share/locale");

    printf(gettext("This is the first string in the first set\n"));
    printf(gettext("and here is the second string in the first set.\n"));
    printf(
        gettext("Now let's have a look at the 1st string in 2nd set,\n"));
    printf(gettext("and finaly the second string in the second set.\n"));
    return EXIT_SUCCESS;
}
```

On voit que l'impact sur le programme est beaucoup plus limité qu'avec `catgets()`. L'application peut être compilée et utilisée immédiatement sans avoir à définir des constantes symboliques. Seules deux lignes ont été ajoutées en début de programme. Quant aux appels `gettext()`, ils pourraient être rendus encore plus discrets à l'aide d'une macro.

Nous créons un fichier `.po` de traduction en français :

exemple_gettext.po :

```
msgid  "This is the first string in the first set\n"
msgstr "Ceci est la première chaîne du premier ensemble \n"

msgid  "and here is the second string in the first set.\n"
msgstr "et voici la seconde chaîne du premier ensemble. \n"

msgid  "Now let's have a look at the 1st string in 2nd set,\n"
msgstr "À présent regardons la 1ère chaîne du 2ème ensemble, \n"

msgid  "and finaly the second string in the second set.\n"
msgstr "et finalement la seconde chaîne du second ensemble. \n"
```

Ce fichier est construit avec des séquences successives utilisant le mot-clé `msgid` pour indiquer la chaîne originale et `msgstr` pour sa traduction.

Nous allons compiler le fichier `.po`, puis nous l'installerons dans le répertoire système d'internationalisation :

```
$ export LC_ALL=fr_FR
$ ./exemple_gettext
This is the first string in the first set
and here is the second string in the first set.
Now let's have a look at the 1st string in 2nd set,
and finaly the second string in the second set.
$ msgfmt -o exemple_gettext.mo exemple_gettext.po
$ su
Password:
# cp exemple_gettext.mo /usr/share/locale/fr/LC_MESSAGES/
# exit
exit
$ ./exemple_gettext
```

```
Ceci est la première chaîne du premier ensemble
et voici la seconde chaîne du premier ensemble.
À présent regardons la 1ère chaîne du 2ème ensemble,
et finalement la seconde chaîne du second ensemble.
$ unset LC_ALL
$ ./exemple_gettext
This is the first string in the first set
and here is the second string in the first set.
Now let's have a look at the 1st string in 2nd set,
and finaly the second string in the second set.
$
```

On remarque que le fichier a été copié dans le répertoire de la localisation `fr` alors que la variable `LC_ALL` a été configurée avec `fr_FR`. La bibliothèque *GetText* recherche en effet intelligemment les fichiers de traduction disponibles.

Il est ainsi possible, de manière simple, de traduire facilement les messages d'interface d'un logiciel. Le projet *GetText* contient également des utilitaires permettant d'analyser le fichier source d'une application existante, afin d'en extraire les chaînes à traduire. Le fichier `.po` est alors construit automatiquement, et il ne reste plus qu'à le soumettre à un traducteur.

Il ne faut toutefois pas oublier que la traduction des messages n'est qu'une partie de l'internationalisation d'un logiciel. De nombreuses conventions culturelles sont parfois aussi importantes que la langue utilisée pour l'interface utilisateur. Si une application affiche le libellé

```
This message was received on 03.04.2000
```

et qu'on traduit uniquement le texte, obtenant ainsi

```
Ce message a été reçu le 03.04.2000
```

il y a de fortes chances pour que le lecteur français lise 3 avril au lieu du 4 mars original.

Un logiciel doit donc pouvoir s'adapter aux règles d'usage décrites dans les autres catégories de localisation.

Configuration de la localisation

Pour qu'une application soit sensible à la localisation, elle doit d'abord invoquer la fonction **`setlocale()`**, déclarée dans `<locale.h>` :

```
char * setlocale (int categorie, const char * localisation);
```

Cette routine demande à la bibliothèque C que toutes les fonctions manipulant des données en rapport avec la catégorie précisée en premier argument prennent en compte le fait que l'utilisateur se trouve dans la localisation indiquée en second argument.

La catégorie est mentionnée sous forme d'une constante symbolique ayant le même nom que les variables d'environnement décrites plus haut : `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`, et surtout `LC_ALL`. Il n'y a pas de constante `LANG`, cette variable d'environnement étant utilisée comme valeur par défaut pour toutes les catégories. Il faut noter que dans l'avenir d'autres catégories seront peut-être ajoutées, et qu'il en existe déjà d'autres sur certains systèmes.

La localisation indiquée en second argument peut prendre l'une des formes suivantes :

- Une chaîne de caractères construite sur le même modèle que le contenu des variables d'environnement ci-dessus (par exemple `"fr_FR"`).
- Une chaîne renvoyée par un appel précédent à `setlocale()`, comme nous le décrirons plus bas.
- Les chaînes spéciales `"POSIX"` ou `"C"`, qui demandent à la bibliothèque d'adopter le comportement décrit exactement dans ces standards, sans s'occuper des variations dues à la localisation. Il s'agit en fait d'une commande d'anti-internationalisation, assurant que le programme fournisse partout le même résultat. Nous reviendrons également sur cette option.
- La chaîne vide `""`, qui demande à la bibliothèque d'adopter le comportement adapté à la localisation configurée par l'utilisateur dans ses variables d'environnement.

En fait, c'est bien entendu la dernière forme qui est la plus utilisée. Il me semble d'ailleurs n'avoir jamais invoqué `setlocale()` dans une application avec d'autres arguments que

```
setlocale (LC_ALL, "");
```

qui réclament de la bibliothèque C une adaptation de ses fonctionnalités, dans toutes les catégories, suivant la localisation choisie par l'utilisateur.

Si on n'appelle pas `setlocale()`, le comportement de la bibliothèque C est le même qu'en ayant invoqué `setlocale (LC_ALL, "C")`, qui ne présente donc pas d'intérêt. Par contre, les localisations `"C"` et `"POSIX"` peuvent être utiles pour configurer une catégorie particulière.

Par exemple, une application peut autoriser un utilisateur à employer ses préférences en termes de langage, de présentation de la date ou de classification des caractères accentués, mais imposer que les saisies de nombres réels se fassent en employant le point comme séparateur décimal. Ceci afin de pouvoir relire automatiquement des fichiers de données déjà construits. On emploiera alors une séquence :

```
    setlocale (LC_ALL, "");
    setlocale (LC_NUMERIC, "C");
```

Ce genre de restriction peut aussi s'appliquer à la catégorie `LC_CTYPE` pour les programmes qui s'appuient fortement sur les correspondances entre les caractères Ascii et leurs valeurs numériques.

Le fait de configurer directement une localisation avec une chaîne explicite ne se justifie que si l'application est lancée sans que les variables d'environnement n'aient pu être configurées par le shell (un démon comme `xdm`). On laissera alors l'utilisateur inscrire ses préférences dans un fichier de configuration, et l'application devra invoquer `setlocale()` avec la chaîne indiquée.

Lorsqu'on passe un argument `NULL` en seconde position de `setlocale()`, cette fonction renvoie un pointeur sur une chaîne de caractères décrivant la localisation actuelle pour la catégorie concernée. Cette chaîne peut être copiée pour une utilisation ultérieure, au besoin. Le pointeur est dirigé vers une zone de mémoire statique interne à la bibliothèque, qui risque d'être écrasée par la suite, et qu'il faut donc copier si on désire la conserver. Si on réclame la valeur de `LC_ALL`, la chaîne renvoyée peut prendre différents formats, car elle représente l'ensemble des catégories (qui peuvent être configurées avec des localisations différentes). Cette chaîne n'est pas obligatoirement intelligible – quoique ce soit apparemment le cas avec la GlibC –, mais elle pourra dans tous les cas être réemployée comme second argument d'un appel `setlocale()` ultérieur.

Voici un programme qui va afficher l'état des localisations programmées :

exemple_setlocale.c :

```
    #include <locale.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    setlocale(LC_ALL, "");
    printf("LC_COLLATE  = %s \n", setlocale(LC_COLLATE,  NULL));
    printf("LC_CTYPE    = %s \n", setlocale(LC_CTYPE,    NULL));
    printf("LC_MESSAGES = %s \n", setlocale(LC_MESSAGES, NULL));
    printf("LC_MONETARY = %s \n", setlocale(LC_MONETARY, NULL));
    printf("LC_NUMERIC  = %s \n", setlocale(LC_NUMERIC,  NULL));
    printf("LC_TIME     = %s \n", setlocale(LC_TIME,     NULL));
    printf("LC_ALL      = %s \n", setlocale(LC_ALL,      NULL));
    return EXIT_SUCCESS;
}
```

En fait, l'exécution nous permet de vérifier que la localisation par défaut est `"C"`, et que les localisations respectent la hiérarchie `LANG` < `LC_xxx` < `LC_ALL`.

```
$ unset LANG LC_ALL
$ ./exemple_setlocale
LC_COLLATE  = C
LC_CTYPE    = C
LC_MESSAGES = C
LC_MONETARY = C
LC_NUMERIC  = C
LC_TIME     = C
LC_ALL      = C
$ export LANG=fr_FR
$ ./exemple_setlocale
LC_COLLATE  = fr_FR
LC_CTYPE    = fr_FR
LC_MESSAGES = fr_FR
LC_MONETARY = fr_FR
LC_NUMERIC  = fr_FR
LC_TIME     = fr_FR
LC_ALL      = fr_FR
$ export LC_CTYPE=fr_CA
$ ./exemple_setlocale
LC_COLLATE  = fr_FR
LC_CTYPE    = fr_CA
LC_MESSAGES = fr_FR
LC_MONETARY = fr_FR
LC_NUMERIC  = fr_FR
LC_TIME     = fr_FR
LC_ALL      = LC_CTYPE=fr_CA;LC_NUMERIC=fr_FR;LC_TIME=fr_FR;LC_COLLATE=fr_FR;
LC_MONETARY=fr_FR;LC_MESSAGES=fr_FR
$ export LC_MONETARY=fr_BE
```

```
$ ./exemple_setlocale
LC_COLLATE  = fr_FR
LC_CTYPE    = fr_CA
LC_MESSAGES = fr_FR
LC_MONETARY = fr_BE
LC_NUMERIC  = fr_FR
LC_TIME     = fr_FR
LC_ALL      = LC_CTYPE=fr_CA;LC_NUMERIC=fr_FR;LC_TIME=fr_FR;LC_COLLATE=fr_FR;
LC_MONETARY=fr_BE;LC_MESSAGES=fr_FR
$ export LC_ALL=es_ES
$ ./exemple_setlocale
LC_COLLATE  = es_ES
LC_CTYPE    = es_ES
LC_MESSAGES = es_ES
LC_MONETARY = es_ES
LC_NUMERIC  = es_ES
LC_TIME     = es_ES
LC_ALL      = es_ES
$
```

Localisation et fonctions de bibliothèques

Une fois que la localisation a été définie pour une ou plusieurs catégories, certaines fonctions de bibliothèque modifient leur comportement pour s'adapter aux coutumes en usage chez l'utilisateur. L'application peut continuer à utiliser `printf()`, par exemple pour afficher des réels, mais le symbole employé pour afficher le séparateur décimal sera modifié. Notons que `printf()` ne permet pas de séparer les chiffres par milliers, au contraire de `strfmon()` que nous étudierons plus loin.

Le programme suivant affiche la valeur numérique 2000,01 avec les variations dues à la localisation :

exemple_numeric.c :

```
    #include <locale.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (int argc, char * argv[])
{
    double d = 2000.01;
    setlocale(LC_ALL, "");
    fprintf(stdout, "%.2f\n", d);
    return EXIT_SUCCESS;
}
```

L'exécution nous montre que seul le point décimal est modifié :

```
$ unset LC_ALL LANG LC_NUMERIC
$ ./exemple_numeric
2000.01
```

```
$ export LANG=fr_FR
$ ./exemple_numeric
2000,01
$
```

De nombreuses routines sont affectées par les catégories `LC_CTYPE` et `LC_COLLATE`, qui concernent les caractères manipulés par le programme. Dans le chapitre 15, nous avons examiné le comportement des fonctions `strcasecmp()`, `strcoll()` ou `strxfrm()` vis-à-vis de la localisation, et nous avons remarqué que les résultats pouvaient varier grandement suivant le jeu de caractères utilisé.

Dans le chapitre 25, nous avons observé aussi que plusieurs fonctions de manipulation des dates étaient sensibles à la localisation *via* la catégorie `LC_TIME`. Cela concerne aussi bien l'affichage avec `strftime()` par exemple que la saisie avec `getdate()`. On peut s'en rendre également compte avec le comportement de l'utilitaire `/bin/date` en l'invoquant avec l'argument `"%A %x"`, qui lui demande d'afficher le nom du jour de la semaine, suivi de la date complète.

```
$ unset LANG LC_TIME LC_ALL
$ date +"%A %x"
Wednesday 03/08/00
$ export LC_TIME=fr_FR
$ date +"%A %x"
mercredi 08.03.2000
$
```

Pour afficher des valeurs monétaires, `printf()` n'est pas vraiment adapté car il ne prend pas en compte le symbole de la monnaie du pays ni certaines coutumes comme la séparation des valeurs par milliers. Pour le remplacer dans ce rôle, il existe une fonction nommée **`strfmon()`**, déclarée dans `<monetary.h>` :

```
ssize_t strfmon (char * buffer, size_t taille, char * format, ...);
```

Cette fonction utilise le format indiqué en troisième argument et convertit les données à sa suite, en stockant le résultat dans le premier argument, dont la taille maximale est indiquée en seconde position. Cette routine se comporte donc un peu comme `snprintf()`, mais elle renvoie le nombre de caractères inscrits dans le buffer. La chaîne de format peut comprendre des caractères normaux, qui seront recopiés directement, ou des indicateurs de conversion commençant par le caractère `%`. Les conversions possibles sont :

Code	Type d'argument	Signification
`%i`	`double`	Représentation locale d'une valeur monétaire représentée sous sa forme internationale.
`%n`	`double`	Représentation locale d'une valeur monétaire représentée sous sa forme nationale.
`%%`		Affichage du caractère `%`.

Entre le `%` et le code de conversion peuvent se trouver plusieurs champs :

- un attribut précisant le formatage du nombre ;
- la largeur minimale de la représentation de la valeur ;

- un symbole # suivi de la largeur minimale de la partie entière, sans compter les éventuels séparateurs de milliers ;
- un point suivi du nombre de décimales à afficher.

L'attribut de formatage peut prendre les formes suivantes :

Code	Signification
=<caractère>	Le caractère indiqué sera employé pour remplir les blancs avant le nombre lorsque le résultat est plus petit que la longueur minimale demandée. En général le caractère par défaut, espace, suffit. Parfois, on peut préférer des zéros ou des points.
!	Ne pas afficher le symbole de la monnaie.
^	Ne pas afficher les séparateurs des milliers.
-	Aligner les nombres à gauche plutôt qu'à droite.
+	Les valeurs sont précédées de leur signe, positif ou négatif.
(	Les valeurs négatives sont entourées de parenthèses. Cet attribut ne peut pas être employé en même temps que le précédent.

Le programme suivant va appeler `strfmon()` avec les arguments passés sur sa ligne de commande.

exemple_strfmon.c :

```
    #include <locale.h>
    #include <monetary.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (int argc, char * argv[])
{
    double d;
    char   buffer[80];

    setlocale (LC_ALL, "");
    if ((argc != 3)
     || (sscanf(argv[2], "%lf", & d) != 1)) {
        fprintf(stderr, "%s format valeur \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (strfmon(buffer, 80, argv[1], d) >0)
        fprintf(stdout, "%s\n", buffer);
    return EXIT_SUCCESS;
}
```

Nous allons tout d'abord observer les effets de la localisation sur une valeur donnée :

```
$ unset LC_ALL LC_MONETARY LANG
$ ./exemple_strfmon "%n" 1500
 1500.00
```

```
$ export LC_MONETARY=fr_FR
$ ./exemple_strfmon "%n" 1500
 1 500,00 EUR
$ ./exemple_strfmon "%i" 1500
 1 500,00 EUR
$ export LC_MONETARY=fr_CH
$ ./exemple_strfmon "%n" 1500
Fr. 1 500.00
$ ./exemple_strfmon "%i" 1500
CHF  1 500.00
$ export LC_MONETARY=en_US
$ ./exemple_strfmon "%n" 1500
 $1,500.00
$ ./exemple_strfmon "%i" 1500
 USD 1,500.00
$
```

On remarque la différence entre les représentations nationales et internationales des monnaies, définies par la norme ISO-4217. Le séparateur des milliers varie aussi. Nous allons observer les remplissages en tête des nombres :

```
$ ./exemple_strfmon "%=0#4.2n" 150
 00150,00 EUR
$ ./exemple_strfmon "%=0#4.2n" 1500
 1 500,00 EUR
$ ./exemple_strfmon "%=0#4.2n" 15000
 15 000,00 EUR
$ ./exemple_strfmon "%=0^#4.2n" 150
 0150,00 EUR
$ ./exemple_strfmon "%=0^#4.2n" 1500
 1500,00 EUR
$
```

Nous voyons que lors du remplissage, la largeur de la partie décimale est complétée avec des zéros, y compris l'espace entre les milliers. Ce comportement permet un alignement correct, que la localisation autorise un séparateur de milliers ou non. Nous voyons aussi que la largeur indiquée n'est qu'un minimum, car lors de la troisième exécution, la valeur n'est pas tronquée. Regardons à présent l'effet des indicateurs de signe :

```
$ ./exemple_strfmon "%n" -1500
-1 500,00 EUR
$ ./exemple_strfmon "%+n" -1500
-1 500,00 EUR
$ ./exemple_strfmon "%+n" 1500
 1 500,00 EUR
$ ./exemple_strfmon "%(n" 1500
 1 500,00 EUR
$ ./exemple_strfmon "%(n" -1500
(1 500,00 EUR)
$
```

Nous laisserons le lecteur expérimenter lui-même les différentes possibilités, en précisant que le caractère '!' peut poser des problèmes avec certains shells et qu'il est préférable pour

l'utiliser de basculer sur un interpréteur ne l'employant pas, comme `ksh`, plutôt que de compliquer la chaîne de format pour protéger le caractère :

```
$ ksh
% ./exemple_strfmon "%!n" 1500
 1 500,00
% ./exemple_strfmon "%n" 1500
 1 500,00 EUR
% exit
$
```

En fait, `strfmon()` ne sait manipuler que des valeurs monétaires, tout comme `strftime()` ne traite que des dates et des heures. Si la chaîne finale doit contenir d'autres conversions, on peut se servir de `strfmon()` pour construire la chaîne de format qui sera employée dans `printf()`. En voici un exemple.

exemple_strfmon_2.c :

```
    #include <locale.h>
    #include <monetary.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    int quantite[] = {
        1, 4, 3, 1, 1, 2, 0
    };
    char * reference[] = {
        "ABC", "DEF", "GHI", "JKL", "MNO", "PQR", NULL
    };
    double prix[] = {
        1500, 2040, 560, 2500, 38400, 125, 0
    };
    int i;
    char format[80];
    double total = 0.0;

    setlocale(LC_ALL, "");
    for (i = 0; reference[i] != NULL; i ++) {
        strfmon(format, 80, "%%5s : %#5n x %%d = %#5n\n",
                prix[i], prix[i] * quantite[i]);
        fprintf(stdout, format, referenc [i], quantite[i]);
        total += prix[i] * quantite[i];
    }

    strfmon(format, 80, "                  Total = %#5n\n", total);
    fprintf(stdout, format);
    return EXIT_SUCCESS;
}
```

La chaîne de format transmise à `fprintf()` permet d'afficher un libellé et un nombre de pièces.

```
$ unset LC_ALL LC_MONETARY LANG
$ export LC_MONETARY=fr_FR
$ ./exemple_strfmon_2
  ABC :   1 500,00€ x 1 =   1 500,00€
  DEF :   2 040,00€ x 4 =   8 160,00€
  GHI :     560,00€ x 3 =   1 680,00€
  JKL :   2 500,00€ x 1 =   2 500,00€
  MNO :  38 400,00€ x 1 =  38 400,00€
  PQR :     125,00€ x 2 =     250,00€
                 Total =  52 490,00€
$ export LC_MONETARY=en_GB
$ ./exemple_strfmon_2
  ABC :  £ 1,500.00 x 1 =  £ 1,500.00
  DEF :  £ 2,040.00 x 4 =  £ 8,160.00
  GHI :  £   560.00 x 3 =  £ 1,680.00
  JKL :  £ 2,500.00 x 1 =  £ 2,500.00
  MNO :  £38,400.00 x 1 =  £38,400.00
  PQR :  £   125.00 x 2 =  £   250.00
                 Total =  £52,490.00
$
```

Localisation et fonctions personnelles

Il peut arriver qu'une application ait à construire elle-même la représentation locale d'une valeur numérique ou monétaire, sans que la fonction `strfmon()` soit suffisante. On peut par exemple avoir besoin de connaître le symbole monétaire employé localement pour l'afficher en tête de colonne d'une facture.

Informations numériques et monétaires avec localeconv()

Il existe une fonction nommée **`localeconv()`**, définie comme `setlocale()` par le standard Iso C, et déclarée dans `<locale.h>` ainsi :

```
struct lconv * localeconv (void);
```

Cette routine renvoie un pointeur sur une zone de données statiques, interne à la bibliothèque, susceptible d'être modifiée, et qu'il ne faut pas écraser. La structure `lconv` renvoyée contient les membres suivants :

Nom	Type	Signification
`decimal_point`	`char *`	Chaîne contenant le caractère employé comme séparateur décimal. Par défaut, il s'agit du point.
`thousands_sep`	`char *`	Chaîne comportant le caractère employé comme séparateur des milliers. Par défaut, la chaîne est vide.
`mon_decimal_point`	`char *`	Comme `decimal_point`, mais appliqué uniquement aux valeurs monétaires.
`mon_thousands_sep`	`char *`	Comme `thousands_sep` pour les valeurs monétaires.
`currency_symbol`	`char *`	Symbole monétaire pour des échanges nationaux.

Nom	Type	Signification
int_curr_symbol	char *	Symbole monétaire pour des échanges internationaux. Conforme à la norme ISO-4217.
positive_sign	char *	Signe employé pour les valeurs monétaires positives. Par défaut, cette chaîne est vide.
negative_sign	char *	Signe utilisé pour les valeurs monétaires négatives. Si cette chaîne est vide, comme c'est le cas par défaut, et si le membre précédent est également vide, il faut employer "-".
frac_digits	char	Nombre de décimales à afficher dans une représentation monétaire nationale. La valeur par défaut, CHAR_MAX, signifie que le comportement n'est pas précisé.
int_frac_digits	char	Nombre de décimales à afficher dans une représentation monétaire internationale. La valeur par défaut, CHAR_MAX, signifie que le comportement n'est pas précisé.
p_cs_precedes	char	Ce membre vaut 1 si le symbole monétaire doit précéder une valeur positive, 0 s'il doit la suivre, et CHAR_MAX si le comportement n'est pas précisé.
p_sep_by_space	char	Ce membre vaut 1 si le symbole monétaire doit être séparé d'une valeur positive par un espace, 0 sinon, et CHAR_MAX si le comportement n'est pas précisé.
p_sign_posn	char	Ce champ peut prendre les valeurs suivantes : 0 si aucun signe n'est affiché devant une valeur positive ; 1 si le signe doit précéder une valeur positive et son symbole ; 2 si le signe doit suivre la valeur positive et le symbole ; 3 si le signe doit se trouver immédiatement avant le symbole ; 4 si le signe doit se trouver immédiatement après le symbole.
n_cs_precedes	char	Comme p_cs_precedes, pour une valeur négative.
n_sep_by_space	char	Comme p_sep_by_space, pour une valeur négative.
n_sign_posn	char	Ce champ est équivalent à p_sign_posn pour des valeurs négatives, mais s'il vaut 0, la valeur négative et son symbole doivent être encadrés par des parenthèses.

Le programme suivant affiche les informations de la structure lconv correspondant à la localisation en cours.

exemple_localeconv.c :

```
    #include <locale.h>
    #include <stdio.h>
    #include <stdlib.h>

    int
main (void)
{
    struct lconv * lconv;

    setlocale(LC_ALL, "");
    lconv = localeconv();
```

```
    printf("decimal_point     = %s \n", lconv->decimal_point);
    printf("thousands_sep     = %s \n", lconv->thousands_sep);
    printf("mon_decimal_point = %s \n", lconv->mon_decimal_point);
    printf("mon_thousands_sep = %s \n", lconv->mon_thousands_sep);
    printf("currency_symbol   = %s \n", lconv->currency_symbol);
    printf("int_curr_symbol   = %s \n", lconv->int_curr_symbol);
    printf("positive_sign     = %s \n", lconv->positive_sign);
    printf("negative_sign     = %s \n", lconv->negative_sign);
    printf("frac_digits       = %d \n", lconv->frac_digits);
    printf("int_frac_digits   = %d \n", lconv->int_frac_digits);
    printf("p_cs_precedes     = %d \n", lconv->p_cs_precedes);
    printf("p_sep_by_space    = %d \n", lconv->p_sep_by_space);
    printf("p_sign_posn       = %d \n", lconv->p_sign_posn);
    printf("n_cs_precedes     = %d \n", lconv->n_cs_precedes);
    printf("n_sep_by_space    = %d \n", lconv->n_sep_by_space);
    printf("n_sign_posn       = %d \n", lconv->n_sign_posn);
    return EXIT_SUCCESS;
}
```

Rappelons qu'une valeur `CHAR_MAX` (127) dans un champ de type `char` signifie que l'information n'est pas disponible.

```
$ unset LC_ALL LANG
$ ./exemple_localeconv
decimal_point     = .
thousands_sep     =
mon_decimal_point =
mon_thousands_sep =
currency_symbol   =
int_curr_symbol   =
positive_sign     =
negative_sign     =
frac_digits       = 127
int_frac_digits   = 127
p_cs_precedes     = 127
p_sep_by_space    = 127
p_sign_posn       = 127
n_cs_precedes     = 127
n_sep_by_space    = 127
n_sign_posn       = 127
$ export LC_ALL=fr_FR
$ ./exemple_localeconv
decimal_point     = ,
thousands_sep     =
mon_decimal_point = ,
mon_thousands_sep =
currency_symbol   = EUR
int_curr_symbol   = EUR
positive_sign     =
negative_sign     = -
frac_digits       = 2
```

```
int_frac_digits   = 2
p_cs_precedes     = 0
p_sep_by_space    = 1
p_sign_posn       = 1
n_cs_precedes     = 0
n_sep_by_space    = 1
n_sign_posn       = 1
$ export LC_ALL=en_US
$ ./exemple_localeconv
decimal_point     = .
thousands_sep     = ,
mon_decimal_point = .
mon_thousands_sep = ,
currency_symbol   = $
int_curr_symbol   = USD
positive_sign     =
negative_sign     = -
frac_digits       = 2
int_frac_digits   = 2
p_cs_precedes     = 1
p_sep_by_space    = 0
p_sign_posn       = 1
n_cs_precedes     = 1
n_sep_by_space    = 0
n_sign_posn       = 1
$
```

Informations complètes avec nl_langinfo()

Il apparaît à l'usage que la fonction `localeconv()` n'est pas suffisante pour obtenir toutes les informations pertinentes concernant la localisation. La limitation aux valeurs numériques et monétaires est très restrictive par rapport au contenu complet des données localisées. Impossible en effet d'avoir accès aux noms des mois sans passer par `strftime()`, ou encore de vérifier si la réponse d'un utilisateur est affirmative ou négative.

Une autre fonction a donc été définie, un peu moins portable que `localeconv()` car elle n'est pas dans la norme ISO C. Nommée **`nl_langinfo()`**, elle se trouve quand même dans les spécifications Unix 98. Sa déclaration se trouve dans `<langinfo.h>` :

```
char * nl_langinfo (nl_item objet);
```

Le type **`nl_item`** est numérique, il est défini dans `<nl_types.h>`. Cette routine renvoie un pointeur sur une chaîne de caractères contenant la représentation locale de l'objet demandé. Contrairement à `localeconv()`, la fonction `nl_langinfo()` permet donc de réclamer uniquement les informations qui nous intéressent.

La chaîne de caractères renvoyée se trouve dans une zone de mémoire statique, susceptible d'être écrasée à chaque appel. L'argument de cette routine est une valeur numérique, qu'on choisit parmi les constantes symboliques suivantes, définies dans `<langinfo.h>` :

Nom	Catégorie	Signification
YESEXPR, NOEXPR	LC_MESSAGES	Chaîne représentant une réponse affirmative ou négative.
MON_DECIMAL_POINT	LC_MONETARY	Significations identiques à celles des champs ayant les mêmes noms dans la structure `lconv` fournie par `localeconv()`.
MON_THOUSANDS_SEP		
CURRENCY_SYMBOL		
INT_CUR_SYMBOL		
POSITIVE_SIGN		
NEGATIVE_SIGN		
FRAC_DIGITS		
INT_FRAC_DIGITS		
P_CS_PRECEDES		
P_SEP_BY_SPACE		
P_SIGN_POSN		
N_CS_PRECEDES		
N_SEP_BY_SPACE		
N_SIGN_POSN		
DECIMAL_POINT	LC_NUMERIC	
THOUSANDS_SEP		
ABDAY_1 ... ABDAY_7	LC_TIME	Abréviations des noms des jours de la semaine.
ABMON_1 ... ABMON_12		Abréviations des noms des mois.
DAY_1 ... DAY_7		Noms des jours de la semaine.
MON_1 ... MON_12		Nom des mois.
AM_STR, PM_STR		Chaînes représentant les symboles AM et PM.
D_FMT, D_T_FMT		Formats pour `strftime()` afin d'obtenir la date seule, ou la date et l'heure.
T_FMT, T_FMT_AMPM		Formats pour avoir l'heure, éventuellement avec les symboles AM et PM.

Il existe quelques autres objets dans la catégorie `LC_TIME`, si la localisation supporte un second calendrier. Ceci est rarement utilisé, et on laissera le lecteur se reporter à la documentation Gnu s'il a besoin de ces fonctionnalités.

Le programme suivant affiche le contenu des champs qui n'étaient pas définis dans la structure `lconv`.

exemple_nl_langinfo.c :

```
    #include <langinfo.h>
    #include <locale.h>
    #include <stdio.h>

    int
main (void)
{
```

```
    int     i;
    char *  libelles [] = {
            "YESEXPR", "NOEXPR",
            "ABDAY_1", "ABDAY_7",
            "ABMON_1", "ABMON_12",
            "DAY_1",   "DAY_7",
            "MON_1",   "MON_12",
            "AM_STR",  "PM_STR",
            "D_FMT",   "D_T_FMT",
            "T_FMT",   "T_FMT_AMPM",
            NULL
    };
    nl_item objet [] = {
            YESEXPR, NOEXPR,
            ABDAY_1, ABDAY_7,
            ABMON_1, ABMON_12,
            DAY_1,   DAY_7,
            MON_1,   MON_12,
            AM_STR,  PM_STR,
            D_FMT,   D_T_FMT,
            T_FMT,   T_FMT_AMPM,
            0
    };
    setlocale (LC_ALL, "");
    for (i = 0; libelles [i] != NULL; i ++)
        fprintf (stdout, "%10s = \"%s\" \n",
                         libelles [i], nl_langinfo (objet [i]));
    return (0);
}
```

Nous n’affichons pas tous les jours de la semaine ni tous les mois.

```
$ unset LC_ALL LANG
$ ./exemple_nl_langinfo
   YESEXPR = "^[yY]"
    NOEXPR = "^[nN]"
   ABDAY_1 = "Sun"
   ABDAY_7 = "Sat"
   ABMON_1 = "Jan"
  ABMON_12 = "Dec"
     DAY_1 = "Sunday"
     DAY_7 = "Saturday"
     MON_1 = "January"
    MON_12 = "December"
    AM_STR = "AM"
    PM_STR = "PM"
     D_FMT = "%m/%d/%y"
   D_T_FMT = "%a %b %e %H:%M:%S %Y"
     T_FMT = "%H:%M:%S"
T_FMT_AMPM = "%I:%M:%S %p"
$
$ export LC_ALL=fr_FR
$ ./exemple_nl_langinfo
```

```
    YESEXPR = "^[oOyY].*"
     NOEXPR = "^[nN].*"
    ABDAY_1 = "dim"
    ABDAY_7 = "sam"
    ABMON_1 = "jan"
   ABMON_12 = "déc"
       DAY_1 = "dimanche"
       DAY_7 = "samedi"
       MON_1 = "janvier"
      MON_12 = "décembre"
      AM_STR = ""
      PM_STR = ""
       D_FMT = "%d.%m.%Y"
    D_T_FMT = "%a %d %b %Y %T %Z"
       T_FMT = "%T"
T_FMT_AMPM = ""
$
```

Nous voyons ainsi qu'on peut obtenir toutes les informations pertinentes concernant les différentes catégories de localisation susceptibles d'être employées dans une application.

Conclusion

Avec les possibilités de traduction des messages d'interface, en employant la bibliothèque Gnu *GetText* et en s'appuyant sur les fonctionnalités offertes par `nl_langinfo()`, une application peut prétendre à une véritable portée internationale.

La documentation Gnu, principalement celle de la bibliothèque *GetText*, offre des informations complémentaires qui intéresseront les développeurs confrontés à des situations plus complexes que celles qui ont été décrites ici.

Ayant examiné depuis quelques chapitres les manipulations possibles sur les données fournies par le système, nous allons changer totalement de sujet, en abordant une nouvelle partie consacrée à l'ensemble des possibilités de communication entre les processus, y compris la programmation réseau.

28

Communications classiques entre processus

Dès qu'une application dépasse un certain degré de complexité pour ce qui concerne les fonctionnalités indépendantes, on peut être tenté de la scinder en plusieurs entités distinctes, sous forme de processus par exemple.

Prenons le cas d'une base de données offrant des possibilités de consultation par l'intermédiaire de connexions TCP/IP. On peut diviser cette application en plusieurs tâches indépendantes. Le noyau principal s'occupe de superviser la base de données elle-même, en gérant notamment les problèmes d'accès simultanés. Un second module assure l'écoute des demandes de connexion et leurs initialisations. Enfin, on peut imaginer disposer d'une multitude de copies d'un dernier module, chargé du déroulement complet de la liaison avec le client, y compris l'interface de dialogue.

Pour construire ce genre de système, plusieurs options se présentent :

- Un seul processus s'occupe de tous les travaux. On conserve en mémoire une copie des données nécessaires au suivi de la connexion pour chaque client. Le processus bascule d'une fonction à l'autre au gré des requêtes grâce à l'appel-système `select()` que nous étudierons dans un prochain chapitre.
- On utilise un système à base de threads, l'accès aux informations globales de la base de données devant être strictement régi par des mutex. Les données propres à chaque connexion sont conservées dans des variables locales de la routine centrale du thread de communication.
- On scinde l'application en plusieurs processus, le noyau principal restant à l'écoute des requêtes de ses fils. Chaque module de communication est représenté par un processus indépendant doté de ses données propres, dialogue avec le client sur une liaison réseau et avec le noyau central par l'intermédiaire de l'une des différentes méthodes que nous allons étudier dans ce chapitre.

Finalement, chacune de ces méthodes a des avantages et des défauts :

- Le processus unique est plus facilement portable sur d'autres systèmes qu'Unix mais, en contrepartie, l'écriture et la maintenance de cette application sont plus compliquées car des fonctionnalités sans rapport entre elles sont regroupées dans le même logiciel.
- La combinaison de plusieurs threads offre une grande souplesse et une bonne portabilité, mais l'indépendance des modules n'est qu'illusoire. Lors d'une évolution du logiciel initial, l'accès à des données globales peut engendrer subitement des bogues difficiles à découvrir.
- La division en plusieurs processus permet d'avoir des modules vraiment indépendants, devant simplement se plier à une interface bien définie. Par contre, le système est dépendant de l'architecture Unix, et la création d'un nouveau processus pour chaque connexion peut parfois être pénalisante.

Dans la première partie de ce chapitre nous allons examiner le moyen de communication le plus simple pour deux processus issus de la même application (père et fils, ou frères) : les tubes.

Il y a également des cas où l'ensemble applicatif repose sur plusieurs logiciels totalement indépendants. Ces programmes doivent disposer d'un autre moyen de communication puisque les tubes ne leur sont plus adaptés. Linux offre alors le concept de tubes nommés, qui sont conçus pour cette situation, et que nous observerons en seconde partie.

Nous nous limitons pour le moment aux communications entre deux processus résidant dans le même système. Lorsqu'on veut faire dialoguer des logiciels se trouvant sur des stations différentes, il faut employer des méthodes que nous examinerons dans les chapitres traitant de la programmation réseau (mais qui ne diffèrent par ailleurs pas beaucoup des principes étudiés ici).

Les tubes

Un tube de communication est un tuyau dans lequel un processus écrit des données qu'un autre processus peut lire. Le tube est créé par l'appel-système **`pipe()`**, déclaré dans `<unistd.h>` :

```
int pipe (int descripteur [2]);
```

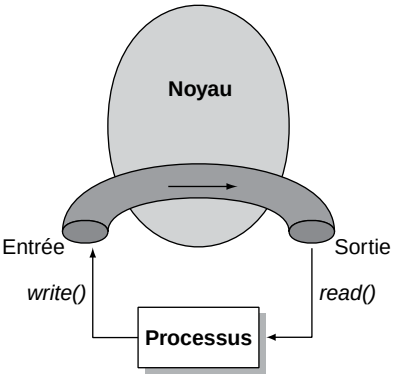
Lorsqu'elle réussit, cette fonction crée un nouveau tube au sein du noyau et remplit le tableau passé en argument avec les descripteurs des deux extrémités. Étant donné que le langage C passe les arguments du type tableau par référence, la routine `pipe()` reçoit un pointeur sur la table et peut donc écrire dans les deux emplacements réservés. Les descripteurs correspondent respectivement à la sortie et à l'entrée du tube.

La situation est résumée sur la figure 28.1.

Le tube est entièrement sous le contrôle du noyau. Il réside en mémoire (et pas sur le disque), et le processus reçoit les deux descripteurs correspondant à l'**entrée** et à la **sortie** du tube. Le descripteur d'indice 0 est la sortie du tube, il est ouvert en lecture seule. Le descripteur 1 est l'entrée ouverte en écriture seule.

Nous observons en effet que les tubes sont des systèmes de communication unidirectionnels. Si on désire obtenir une communication complète entre deux processus, il faut créer deux tubes et les employer dans des sens opposés.

Figure 28.1
Tube de communication



Dans notre premier exemple, nous allons simplement créer un tube, écrire des données dedans, lire son contenu et vérifier que les informations sont identiques.

exemple_pipe_1.c :

```
    #include <stdio.h>
    #include <unistd.h>

    int
main (void)
{
    int             tube[2];
    unsigned char   buffer[256];
    int             i;

    fprintf(stdout, "Création tube \n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < 256; i ++)
        buffer[i] = i;
    fprintf(stdout, "Écriture dans tube \n");
    if (write(tube[1], buffer, 256) != 256) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Lecture depuis tube \n");
    if (read(tube[0], buffer, 256) != 256) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Vérification...");
    for (i = 0; i < 256; i ++)
        if (buffer[i] != i) {
            fprintf(stdout, "Erreur : i=%d buffer[i]=%d \n",
                                i, buffer[i]);
```

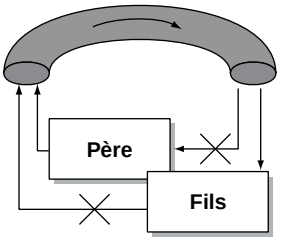
```
            exit(EXIT_FAILURE);
        }
    fprintf(stdout, "Ok \n");
    return EXIT_SUCCESS;
}
```

Nous vérifions son fonctionnement :

```
$ ./exemple_pipe_1
Création tube
Écriture dans tube
Lecture depuis tube
Vérification...Ok
$
```

Utiliser un tube pour transférer des données au sein du même processus ne présente aucun intérêt. Aussi nous allons utiliser ce mécanisme pour faire communiquer deux processus (ou plus). Pour cela, nous devons invoquer l'appel-système `fork()` après avoir créé le tube. Si celui-ci doit aller du processus père vers le fils, le père ferme son descripteur de sortie de tube, et le fils son descripteur d'entrée. Nous expliquerons plus bas pourquoi la fermeture des extrémités inutilisées est importante. La figure 28.2 présente cet état de fait.

Figure 28.2
Tube du père vers le fils



Notre second exemple permet de tester ceci.

exemple_pipe_2.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/wait.h>

    int
main (void)
{
    int             tube[2];
    unsigned char   buffer[256];
    int             i;

    fprintf(stdout, "Création tube \n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
```

```
    switch (fork()) {
        case -1 :
            perror("fork()");
            exit(EXIT_FAILURE);
        case 0 :
            fprintf(stdout, "Fils : Fermeture entrée \n");
            close(tube[1]);
            fprintf(stdout, "Fils : Lecture tube \n");
            if (read(tube[0], buffer, 256) != 256) {
                perror("read");
                exit(EXIT_FAILURE);
            }
            fprintf(stdout, "Fils : Vérification \n");
            for (i = 0; i < 256; i ++)
                if (buffer[i] != i) {
                    fprintf(stdout, "Fils : Erreur \n");
                    exit(EXIT_FAILURE);
                }
            fprintf(stdout, "Fils : Ok \n");
            break;
        default :
            fprintf(stdout, "Père : Fermeture sortie \n");
            close(tube[0]);
            for (i = 0; i < 256; i ++)
                buffer[i] = i;
            fprintf(stdout, "Père : Écriture dans tube \n");
            if (write(tube[1], buffer, 256) != 256) {
                perror("write");
                exit(EXIT_FAILURE);
            }
            wait(NULL);
            break;
    }
    return EXIT_SUCCESS;
}
```

L'exécution confirme bien le fonctionnement du tube allant du processus père vers son fils.

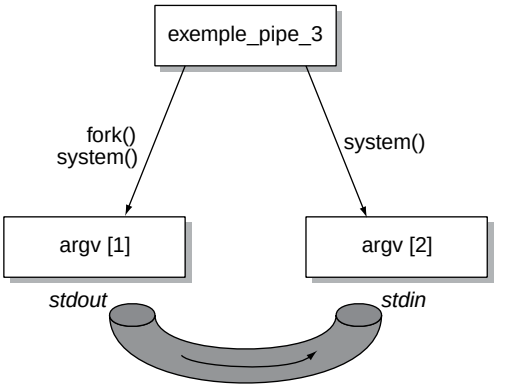
```
$ ./exemple_pipe_2
Création tube
Père : Fermeture sortie
Fils : Fermeture entrée
Fils : Lecture tube
Père : Écriture dans tube
Fils : Vérification
Fils : Ok
$
```

Nous remarquons que ce système est semblable au principe du *pipe* des shells, qui permet grâce au caractère « | » de diriger la sortie standard d'un processus vers l'entrée standard d'un autre. Pour illustrer ce principe, nous allons créer un programme qui prend deux commandes en arguments et qui les exécute en redirigeant la sortie standard de la première vers un tube connecté à l'entrée standard de la seconde.

Pour lancer les commandes nous utilisons `fork()` pour dissocier deux processus, le fils exécutant la première commande, et le père la seconde. Pour éviter d'avoir à analyser les chaînes de caractères pour séparer les commandes de leurs arguments, nous faisons appel à la fonction `system()`. En fait, il aurait été plus élégant d'employer `execvp()`, mais le traitement des chaînes de commande aurait été plus compliqué.

Le principe de notre programme est illustré par la figure 28.3.

Figure 28.3
Création d'un tube entre deux commandes



Nous utilisons l'appel-système `dup2()` que nous avons décrit dans le chapitre 19 pour remplacer les flux `stdin` et `stdout` des processus par les extrémités du tube.

exemple_pipe_3.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (int argc, char * argv[])
{
    int  tube[2];

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s commande_1 commande_2\n",
                            argv[0]);
        exit(EXIT_FAILURE);
    }
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    switch (fork()) {
        case -1 :
            perror("fork()");
            exit(EXIT_FAILURE);
```

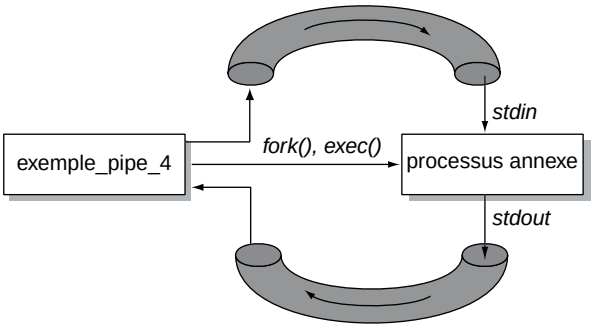
```
        case 0 :
            close(tube[0]);
            dup2(tube[1], STDOUT_FILENO);
            system(argv[1]);
            break;
        default :
            close(tube[1]);
            dup2(tube[0], STDIN_FILENO);
            system(argv[2]);
            break;
    }
    return EXIT_SUCCESS;
}
```

Pour vérifier notre programme, nous allons lui faire exécuter l'équivalent de la commande shell `ls -l /dev | grep cdrom` :

```
$ ./exemple_pipe_3 "ls -l /dev" "grep cdrom"
lrwxrwxrwx   1 root     root       3 Aug 12 17:57 cdrom -> hdc
$
```

Nous voyons ainsi un mécanisme de plus employé par les interpréteurs de commande pour implémenter toutes les fonctionnalités qu'ils offrent. Nous allons améliorer encore notre programme en implémentant une possibilité rarement proposée par les shells. Notre processus va rediriger l'entrée et la sortie standard d'un programme qu'il exécute. Ceci permet d'utiliser une autre application comme une sous-routine du programme. Le principe est un peu semblable à celui de `popen()`, mais cette fonction ne pouvait rediriger que l'entrée ou la sortie du processus appelé, alors que nous allons traiter les deux simultanément. Pour garder un exemple assez simple, nous invoquerons l'utilitaire `wc` qui peut compter le nombre de caractères, de mots ou de lignes dans un fichier de texte. Nous allons lui transmettre le contenu complet d'un fichier sur son entrée standard et lire sa réponse sur sa sortie standard. La figure 28.4 illustre cet exemple.

Figure 28.4
Utilisation d'un processus annexe



exemple_pipe_4.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
```

```
    int
invoque_processus (const char * commande, int fd[2])
{
    int  tube_1[2];
    int  tube_2[2];
    if ((pipe(tube_1) != 0) || (pipe(tube_2) != 0))
        return -1;
    switch (fork()) {
        case -1 :
            close(tube_1[0]); close(tube_1[1]);
            close(tube_2[0]); close(tube_2[1]);
            return -1;
        case 0 :
            close(tube_1[1]);
            close(tube_2[0]);
            dup2(tube_1[0], STDIN_FILENO);
            dup2(tube_2[1], STDOUT_FILENO);
            system(commande);
            exit(EXIT_SUCCESS);
        default :
            close(tube_1[0]);
            close(tube_2[1]);
            fd[0] = tube_2[0];
            fd[1] = tube_1[1];
    }
    return 0;
}

    int
main (int argc, char * argv[])
{
    int    tube[2];
    FILE * fichier;
    char * contenu;
    char   c;
    struct stat status;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (stat(argv[1], & status) != 0) {
        perror("stat");
        exit(EXIT_FAILURE);
    }
    if ((contenu = malloc(status.st_size)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    if ((fichier = fopen(argv[1], "r")) == NULL) {
        perror("fopen");
```

```
        exit(EXIT_FAILURE);
    }
    if (fread(contenu, 1, status.st_size, fichier) !=
                        status.st_size) {
        perror("fread");
        exit(EXIT_FAILURE);
    }
    fclose(fichier);
    if (invoque_processus("wc -w", tube) != 0) {
        perror("invoque_processus");
        exit(EXIT_FAILURE);
    }
    write(tube[1], contenu, status.st_size);
    close(tube[1]);
    fprintf(stdout, "Nombre de mots :   ");
    while (read(tube[0], & c, 1) == 1)
        fputc(c, stdout);
    close(tube[0]);

    if (invoque_processus("wc -l", tube) != 0) {
        perror("invoque_processus");
        exit(EXIT_FAILURE);
    }
    write(tube[1], contenu, status.st_size);
    close(tube[1]);
    fprintf(stdout, "Nombre de lignes : ");
    while (read(tube[0], & c, 1) == 1)
        fputc(c, stdout);
    close(tube[0]);

    free(contenu);
    return EXIT_SUCCESS;
}
```

Notre programme appelle successivement `wc -w` pour avoir le nombre de mots, et `wc -l` pour le nombre de lignes.

```
$ ./exemple_pipe_4 exemple_pipe_4.c
Nombre de mots :       298
Nombre de lignes :      93
$
```

On peut noter que nous avons pris soin, lorsque nous avons fini d'écrire toutes nos données dans le tube, de fermer cette extrémité. À ce moment, en effet, le noyau voit qu'il n'y a plus de processus disposant d'un descripteur sur l'entrée du tube puisque nous avions fermé également la copie de ce descripteur dans le processus fils. Dès qu'un processus tentera de lire à nouveau dans le tube, comme le fait l'utilitaire `wc`, le noyau lui enverra le symbole `EOF`, fin de fichier. Il est donc important de bien refermer l'extrémité du tube qu'on n'utilise pas immédiatement après avoir appelé `fork()`.

Symétriquement, au moment d'une tentative d'écriture dans un tube dont tous les descripteurs de sortie ont été fermés, le processus appelant `write()` reçoit le signal `SIGPIPE`.

Par défaut ce signal tue le processus écrivain, comme nous le voyons dans le programme suivant :

exemple_pipe_5.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    int    tube[2];
    char * buffer = "AZERTYUIOP";

    fprintf(stdout, "Création tube \n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Fermeture sortie \n");
    close(tube[0]);
    fprintf(stdout, "Écriture dans tube \n");
    if (write(tube[1], buffer, strlen(buffer)) != strlen(buffer)) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Fin du programme \n");
    return EXIT_SUCCESS;
}
```

Nous observons que le programme est tué avant d'avoir pu écrire son dernier message.

```
$ ./exemple_pipe_5
Création tube
Fermeture sortie
Écriture dans tube
Broken pipe
$
```

Le message « *Broken pipe* » provient du shell qui nous indique ainsi quel signal a tué notre processus. Si nous ignorons le signal `SIGPIPE` ou si nous le capturons, l'appel-système `write()` échoue avec l'erreur `EPIPE`. Le programme `exemple_pipe_6` est une copie de `exemple_pipe_5` avec la ligne suivante ajoutée en début de fonction `main()` :

```
    signal (SIGPIPE, SIG_IGN);
```

L'exécution montre alors que l'échec se produit à présent dans l'appel `write()`.

```
$ ./exemple_pipe_6
Création tube
Fermeture sortie
Écriture dans tube
write: Relais brisé (pipe)
$
```

Nous avons constaté dans nos premiers exemples qu'il est tout à fait possible d'écrire dans un tube sans en lire immédiatement le contenu – à condition qu'un descripteur de sa sortie reste ouvert. Cela signifie donc que le noyau associe une mémoire tampon à chaque tube. La dimension de cette mémoire est représentée par la constante symbolique `PIPE_BUF`, définie dans le fichier `<limits.h>`. Sur un PC sous Linux, cette constante vaut 4096. Nous pouvons le vérifier en écrivant dans un tube un caractère à la fois, en regardant au bout de combien d'octets l'écriture devient bloquante.

exemple_pipe_7.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    int   tube[2];
    char  c = 'c';
    int   i;

    fprintf(stdout, "Création tube \n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Écriture dans tube \n");
    for (i = 0; ; i ++) {
        fprintf(stdout, "%d octets écrits \n", i);
        if (write(tube[1], & c, 1) != 1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
    }
    return EXIT_SUCCESS;
}
```

Lorsque le buffer est plein, l'écriture reste bloquée, et nous interrompons le processus en appuyant sur Contrôle-C :

```
$ ./exemple_pipe_7
0 octets écrits
1 octets écrits
2 octets écrits
   [...]
4092 octets écrits
4093 octets écrits
4094 octets écrits
4095 octets écrits
4096 octets écrits

  (Contrôle-C)
$
```

En fait, la constante `PIPE_BUF` correspond également à la taille maximale d'un bloc de données qui puisse être écrit de manière atomique. Lorsque plusieurs processus partagent un même descripteur sur l'entrée du tube, leurs écritures respectives ne seront pas entremêlées si elles ne dépassent pas `PIPE_BUF` octets à la fois. Le noyau garantit dans ce cas l'atomicité du transfert des données vers le buffer – quitte à bloquer l'écriture avant le transfert s'il n'y a pas assez de place dans la zone tampon.

On peut s'interroger sur les informations renvoyées lorsqu'on invoque l'appel-système `stat()` sur un descripteur de fichier. Nous pouvons en faire l'expérience.

exemple_pipe_8.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/stat.h>

    int
main (void)
{
    int         tube [2];
    struct stat status;

    fprintf(stdout, "Création tube \n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    if (fstat(tube[0], & status) != 0) {
        perror("fstat");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Status : ");
    if (S_ISFIFO(status .st_mode))
        fprintf(stdout, "FIFO\n");
    else
        fprintf(stdout, "? \n");
    return EXIT_SUCCESS;
}
```

Finalement le programme est un peu biaisé car nous devinons déjà ce que nous pouvons attendre.

```
$ ./exemple_pipe_8
Création tube
Status : FIFO
$
```

Le type d'un descripteur de fichier appartenant à un tube est donc « FIFO ». Les autres membres de la structure `stat` sont vides ou sans signification – hormis les champs `st_uid` et `st_gid` qui représentent l'identité du processus ayant créé le tube.

Les tubes obtenus par l'appel-système `pipe()` représentent donc un moyen de communication simple mais très efficace entre des processus différents. Un problème se pose pourtant, car il

faut nécessairement que les interlocuteurs aient un ancêtre commun, le processus qui a créé le tube[1]. Il n'est pas possible de lancer des programmes indépendants – par exemple un serveur et des clients – et qu'ils établissent un dialogue.

Les tubes nommés

Pour permettre ce genre de communications, le concept de tube a été étendu pour disposer d'un nom dans le système de fichiers, donnant naissance au terme peu esthétique de « tube nommé » (*named pipe*).

Un tube nommé est donc simplement un nœud dans le système de fichiers. Lorsqu'on l'ouvre pour la première fois, le noyau crée un tube de communication en mémoire. Chaque écriture et chaque lecture auront donc lieu dans ce tube, avec les mêmes principes que ceux que nous avons étudiés à la section précédente.

Ce moyen de communication disposant d'une représentation dans le système de fichiers, des processus indépendants peuvent l'employer pour dialoguer, sans qu'ils soient obligés d'être tous lancés par la même application. Les processus peuvent même appartenir à des utilisateurs différents.

Par ailleurs, il est fréquent que plusieurs processus clients écrivent dans le même tube nommé, afin qu'un processus serveur lisent les requêtes. Nous écrirons un tel programme plus bas. Ceci est possible aussi avec des tubes simples, mais c'est de plus en plus rare car on préfère dans ce cas créer un canal de communication pour chaque client.

Le nœud du système de fichiers représentant un tube nommé est du type Fifo (*first in first out*), dont nous avons parlé dans le chapitre 21. La création d'un tel nœud peut se faire avec la fonction **`mkfifo()`**, déclarée dans `<sys/stat.h>` :

```
int mkfifo (const char * nom, mode_t mode);
```

Cette fonction renvoie 0 si la création a réussi et –1 en cas d'échec, par exemple si le nœud existait déjà. Le mode indiqué en second argument est identique à celui qui est employé dans l'appel-système `open()`. En fait, cette fonction de bibliothèque invoque directement l'appel-système `mknod()` ainsi :

```
    int
mkfifo (const char * nom, mode_t mode)
{
    dev_t dev = 0;
    return mknod(nom, mode | S_IFIFO, dev);
}
```

Le troisième argument de `mknod()` est ignoré quand le nœud n'est pas un fichier spécial de périphérique.

On peut aussi employer l'utilitaire `/usr/bin/mkfifo`, qui sert de frontal à cette fonction, avec une option `–m` permettant d'indiquer le mode désiré. Une fois le nœud créé, on peut l'ouvrir avec `open()` – avec les restrictions dues au mode d'accès –, écrire dedans avec `write()`, y lire avec `read()`, et le refermer avec `close()`. La suppression d'un tube nommé se fait avec l'appel-système `unlink()`.

1. Bien entendu tous les processus ont un ancêtre commun, `init` de PID 1, et la plupart d'entre eux descendent du même shell. Ce qui nous gêne ici c'est que l'ancêtre doit appartenir à la même application.

Lorsqu'on ouvre un tube nommé en lecture seule, l'appel `open()` reste bloqué jusqu'à ce que le tube soit ouvert en écriture par un autre processus. Parallèlement, une ouverture en écriture seule est bloquante jusqu'à ce que le tube soit ouvert en écriture par un autre processus.

L'ouverture en lecture et écriture n'est pas portable, car même si la plupart des systèmes l'acceptent, SUSv3 précise que ce comportement est indéfini. Pour pouvoir ouvrir simultanément les deux extrémités dans le même processus, on emploie l'attribut `O_NONBLOCK` lors de l'appel-système `open()` afin de permettre une ouverture non bloquante. Dans ce cas, une ouverture en lecture seule n'attendra pas qu'un autre processus ouvre le tube en écriture, et inversement. Nous nous trouvons alors dans la même situation que lorsque le correspondant a fermé son extrémité du tube.

Une lecture depuis un tube non ouvert du côté écriture renverra `EOF`, et une écriture dans un tube dont la sortie est fermée déclenchera `SIGPIPE`. Il faut savoir que l'attribut `O_NONBLOCK` concerne aussi les lectures ou écritures ultérieures. Les appels-système `read()` et `write()` deviennent alors non bloquants, comme nous le verrons dans le 30.

Une ouverture non bloquante, en lecture seule, renverra toujours 0, alors qu'une ouverture non bloquante en écriture seule déclenchera l'erreur `ENXIO`.

Il est théoriquement possible d'employer `fopen()` pour ouvrir une Fifo. Toutefois, je préfère utiliser systématiquement `open()` renvoyant un descripteur, suivi éventuellement d'un `fdopen()` me fournissant un flux à partir du descripteur obtenu, ceci pour plusieurs raisons :

- Lors d'un `fopen()`, la bibliothèque C invoque `open()`, mais nous ne savons pas toujours avec quels arguments. Nous pouvons examiner les sources de la GlibC mais, en cas de portage sur un autre système, nous n'avons pas nécessairement accès aux sources de la bibliothèque C.
- La fonction `fopen()` ne permet pas, contrairement à `open()`, de demander une ouverture non bloquante, ce qui est souvent indispensable, notamment lorsqu'on veut ouvrir les deux extrémités d'un tube nommé dans le même processus.
- L'appel-système `open()` n'autorise pas la création d'un fichier si l'attribut `O_CREAT` n'est pas présent. Au contraire, `fopen()` en mode « `w` » risque de créer un fichier normal si le nœud de la Fifo n'a pas encore été créé par un autre processus. Non seulement les communications ne fonctionneront pas, mais si nous détruisons le fichier en fin de programme avec `unlink()`, nous avons peu de chances de nous rendre compte du problème sans passer par une session de débogage assez pénible.

Dans le programme suivant nous allons utiliser plusieurs tubes nommés pour faire dialoguer un processus serveur avec plusieurs clients. Le serveur crée un nœud dans le système de fichier et y lit les requêtes des clients. Ce tube dispose d'un nom connu par tous les processus. Pour répondre à la requête d'un client, le serveur doit pouvoir écrire dans un autre tube nommé, spécifique au client. Les requêtes doivent avoir une taille inférieure à `PIPE_BUF`, afin d'être sûr qu'elles ne seront pas mélangées dans le tube d'interrogation. Comme il faut bien donner un travail à faire au serveur vis-à-vis des clients, nous allons simplement lui faire renvoyer un anagramme de la chaîne de caractères transmise dans la requête.

Le principe retenu pour faire fonctionner l'ensemble est le suivant :

- Le serveur crée un nœud dans le système de fichiers, nommé `anagramme.fifo`. Il ouvre ensuite ce tube en lecture et en écriture, puis le rouvre sous forme de flux.

- Un client essaye d'ouvrir en écriture seule le tube du serveur. S'il n'existe pas, le processus client se termine. Sinon, le client crée un tube personnel, nommé `anagramme.<pid>`, afin d'être unique dans notre application.
- Le client envoie au serveur une requête constituée du nom du tube pour la réponse, suivi d'un retour chariot et de la chaîne de caractères dont on désire un anagramme, suivie d'un retour chariot. Le serveur peut alors lire grâce à `fgets()` ces éléments et répondre dans le tube du client.
- Le client ouvre son tube en lecture seule, lit la réponse, l'affiche et se termine, après avoir supprimé avec `unlink()` son tube de réponse.
- Si la chaîne de caractères reçue vaut `FIN`, le serveur se termine également en supprimant le tube d'interrogation.

Plusieurs clients peuvent travailler simultanément avec le serveur, car le noyau nous assure que toute requête dont la taille est inférieure à `PIPE_BUF` sera traitée de manière atomique. Si le buffer ne dispose pas d'assez de place pour stocker les données, l'appel `write()` attendra qu'il se libère, mais la copie dans le buffer sera accomplie en une seule fois, sans qu'une autre écriture ne puisse interférer.

Nous avons ouvert, dans le serveur, le tube d'interrogation en lecture et écriture. Ceci nous évite de rester bloqué durant l'ouverture en attendant qu'un autre processus soit prêt à écrire, mais on y trouve aussi un second avantage. Si nous ouvrons le tube en lecture seule, à chaque fois que le client se termine, la lecture dans le serveur avec `fgets()` échoue car le noyau envoie un `EOF`. En demandant un tube en lecture et écriture, nous évitons cette situation, car il reste toujours au moins un descripteur ouvert sur la sortie. Nous supprimons ainsi un cas d'échec possible.

Ce programme doit juste être considéré comme un exemple simpliste pour démontrer les possibilités des tubes nommés ; il lui manque un grand nombre de vérifications des conditions d'erreur.

exemple_serveur.c :

```
    #define _GNU_SOURCE /* Pour strfry() */

    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/types.h>

    static char * nom_noeud = "anagramme.fifo";

    static int
repondre (const char * nom_fifo, char * chaine)
{
    FILE * reponse;
    int    fd;
    char * anagramme;
```

```
    if ((fd = open(nom_fifo, O_WRONLY)) >= 0) {
        reponse = fdopen(fd, "w");
        anagramme = strdup(chaine);
        strfry(anagramme);
        fprintf(reponse, "%s\n", anagramme);
        fclose(reponse);
        free(anagramme);
    }
    if ((strcasecmp(chaine, "FIN") == 0)
     || (strcasecmp(nom_fifo, "FIN") == 0))
        return 1;
    return 0;
}

    int
main (void)
{
    FILE * fichier;
    int    fd;
    char   nom_fifo[128];
    char   chaine[128];

    if (mkfifo(nom_noeud, 0644) != 0) {
        fprintf(stderr, "Impossible de créer le noeud Fifo \n");
        exit(EXIT_FAILURE);
    }
    fd = open(nom_noeud, O_RDWR);
    fichier = fdopen(fd, "r");
    while (1) {
        fgets(nom_fifo, 128, fichier);
        if (nom_fifo[strlen(nom_fifo) - 1] == '\n')
            nom_fifo[strlen(nom_fifo) - 1] = '\0';
        fgets(chaine, 128, fichier);
        if (chaine[strlen(chaine) - 1] == '\n')
            chaine[strlen(chaine) - 1] = '\0';
        if (repondre(nom_fifo, chaine) != 0)
            break;
    }
    unlink(nom_noeud);
    return EXIT_SUCCESS;
}
```

Le processus client est construit ainsi :

exemple_client.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>
    #include <sys/stat.h>
    #include <sys/types.h>
```

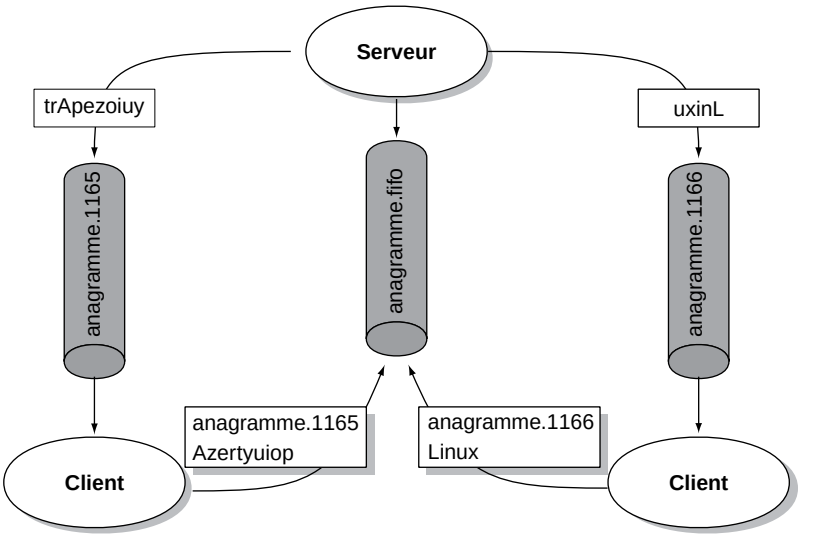
```
    int
main (void)
{
    FILE * question;
    FILE * reponse;
    int    fd;
    char   nom_fifo[128];
    char   chaine[128];

    fprintf(stdout, "Chaîne à traiter : ");
    if (fgets (chaine, 128, stdin) == NULL)
        exit (0);
    sprintf(nom_fifo, "anagramme.%ld", (long) getpid());
    if (mkfifo(nom_fifo, 0644) != 0) {
        fprintf(stderr, "Impossible de créer le noeud Fifo \n");
        exit(EXIT_FAILURE);
    }
    if ((fd = open("anagramme.fifo", O_WRONLY)) < 0) {
        fprintf(stderr, "Impossible d'ouvrir la Fifo \n");
        exit(EXIT_FAILURE);
    }
    question = fdopen(fd, "w");
    fprintf(question, "%s\n%s", nom_fifo, chaine);
    fclose(question);
    fd = open(nom_fifo, O_RDONLY);
    reponse = fdopen(fd, "r");
    if (fgets(chaine, 128, reponse) != NULL)
        fprintf(stdout, "Réponse = %s\n", chaine);
    else
        perror("fgets");
    fclose(reponse);
    unlink(nom_fifo);
    return EXIT_SUCCESS;
}
```

L'exécution suivante est résumée sur la figure 28.5. Le processus serveur est lancé en arrière-plan, le shell nous indique quand il se termine avec la ligne *Done exemple_serveur*.

```
$ ./exemple_serveur &
[1] 1163
$ ls -l anagramme.fifo
prw-r--r--   1 ccb   ccb      0 Jan 31 13:38 anagramme.fifo
$ ./exemple_client
Chaîne à traiter : Azertyuiop
Réponse = trApezoiuy

$ ./exemple_client
Chaîne à traiter : Linux
Réponse = uxinL

$ ./exemple_client
Chaîne à traiter : Anagramme
Réponse = emAngmara
```

```
$ ./exemple_client
Chaîne à traiter : fin
[1]+  Done              ./exemple_serveur
$ ls -l anagramme.fifo
ls: anagramme.fifo: Aucun fichier ou répertoire de ce type
$
```

Figure 28.5
Serveur d'anagramme à base de tubes nommés



On peut remarquer que le type Fifo est indiqué par le caractère `p` (*pipe*) dans l'affichage de la commande `ls -l`. Nous pouvons également écrire directement dans le tube nommé `anagramme.fifo`, à l'aide de la commande `echo` du shell et de son option –`e` qui lui permet d'interpréter le caractère `\n`.

```
$ ./exemple_serveur &
[1] 1170
$ ls -l anagramme.fifo
prw-r--r--   1 ccb   ccb       0 Jan 31 13:39 anagramme.fifo
$ echo -e "FIN\nFIN" > anagramme.fifo
[1]+  Done                    ./exemple_serveur
$ ls -l anagramme.fifo
ls: anagramme.fifo: Aucun fichier ou répertoire de ce type
$
```

J'encourage vivement le lecteur à expérimenter différents cas de figure dans les ouvertures des tubes nommés, afin de bien saisir les points de blocage et les moments où `fgets()` échoue car le correspondant a refermé son extrémité du tube. Pour bien suivre l'état des différents processus concernés, on peut regarder régulièrement le contenu du pseudo-répertoire `/proc/<pid>/fd/`, qui contient des liens symboliques décrivant les divers descripteurs employés par un programme.

Conclusion

Nous avons observé ici une méthode permettant de transférer des données d'un processus à un autre. Dans le cas du tube simple, on est limité à des descendants du processus de démarrage de l'application, mais le tube nommé permet d'étendre ce mécanisme en utilisant un nom dans le système de fichiers.

En combinant les tubes et les signaux, on peut très bien parvenir à une bonne communication entre les processus. Toutefois d'autres mécanismes sont également disponibles, comme nous le verrons dans le prochain chapitre. Nous nous intéresserons aux possibilités de multiplexage d'entrées-sorties dans le chapitre 30, qui sont très utiles dans les communications avec des tubes.

Le principe de fonctionnement des tubes est important car il introduit un concept qu'on développera très largement dans les chapitres 31 et 32 avec les communications réseau.

29

Communications avec les IPC Système V

Ce qu'on nomme généralement *IPC Système V* recouvre trois mécanismes de communication entre processus (*IPC, Inter Process Communication*), apparus en 1983 dans la première version d'Unix Système V, mais qui n'ont été que tardivement intégrés dans les standards comme Unix 98, et n'étaient pas définis par Posix. De nos jours elles sont finalement définies par SUSv3.

Beaucoup de programmeurs rechignent à employer ces méthodes de dialogue, car elles ne sont pas fondées sur le concept des descripteurs de fichiers, contrairement aux tubes, aux tubes nommés ou même aux sockets. Il n'est pas possible d'employer des schémas homogènes pour traiter toutes les entrées-sorties d'un processus ni d'utiliser les méthodes de multiplexage et de traitements asynchrones que nous rencontrerons dans le prochain chapitre. Pourtant, il existe encore de nombreuses applications faisant usage de ces mécanismes, aussi allons-nous les étudier à présent.

On notera tout de suite que les IPC Système V peuvent être autorisés ou non lors de la compilation du noyau Linux. Ainsi, si les appels-système décrits dans ce chapitre échouent toujours avec l'erreur `ENOSYS`. Il faudra donc recompiler le noyau pour pouvoir les utiliser.

Principes généraux des IPC Système V

Les IPC regroupent, on l'a dit, trois méthodes de communication :

- Les files de messages, dans lesquelles un processus peut glisser des données ou en extraire. Les messages étant typés, il est possible de les lire dans un ordre différent de celui d'insertion, bien que par défaut la lecture se fasse suivant le principe d'une file d'attente.
- Les segments de mémoire partagée, qui sont accessibles simultanément par deux processus ou plus, avec éventuellement des restrictions telles que la lecture seule.

- Les sémaphores, qui permettent de synchroniser l'accès à des ressources partagées. Nous avons déjà rencontré les sémaphores Posix.1b dans le chapitre 12, et il ne faut pas les confondre avec les sémaphores des IPC Système V, même si le principe reste globalement le même.

Dans tous les cas, ces outils de communication peuvent être partagés entre des processus n'ayant pas immédiatement d'ancêtre commun. Pour cela, les IPC introduisent le concept de clé.

Obtention d'une clé

Une ressource IPC partagée est accessible par l'intermédiaire d'un nombre entier servant d'identificateur – numéro de la file de messages ou de l'ensemble de sémaphores, identifiant du segment mémoire –, qui est commun aux processus désirant l'utiliser.

Pour partager ce numéro d'identification, un processus peut demander au système de créer la ressource de manière privée, puis transmettre directement le numéro – par l'intermédiaire d'un fichier par exemple – aux autres processus avec lesquels il veut communiquer. Ce schéma est souvent utilisé si un processus ne désire communiquer qu'avec ses descendants, car il crée alors la ressource IPC privée avant d'invoquer `fork()`.

Il est aussi possible de se servir, lors de la demande de création de ressource, d'une clé qui permettra au système d'identifier l'IPC désiré. Cette clé peut être commune à plusieurs processus, qui se mettent d'accord pour employer une valeur figée, un peu à la manière des numéros de ports lors des connexions réseau. Il faut alors documenter proprement l'application pour bien indiquer les clés qu'elle utilise.

Une dernière possibilité, et finalement la meilleure, consiste à demander au système de créer lui-même une clé, fondée sur des références communes pour tous les processus. La clé est constituée en employant un nom de fichier et un identificateur de projet. De cette manière, tous les processus d'un ensemble donné pourront choisir de créer leur clé commune en utilisant le chemin d'accès du fichier exécutable de l'application principale, ainsi qu'un numéro de version par exemple.

Une clé est fournie par le système sous forme d'un objet de type **`key_t`**, défini dans `<sys/type.h>`. La constante symbolique **`IPC_PRIVATE`**, définie dans `<sys/ipc.h>` représente une clé privée, demandant sans condition la création d'une nouvelle ressource IPC, comme dans le premier schéma que nous avons imaginé.

Pour créer une nouvelle clé à partir d'un nom de fichier et d'un identificateur de projet, on emploie la fonction **`ftok()`**, déclarée ainsi dans `<sys/ipc.h>` :

```
key_t ftok (char * nom_fichier, char projet);
```

La clé créée emploie une partie du numéro d'i-nœud du fichier indiqué, le numéro mineur du périphérique sur lequel il se trouve et la valeur transmise en second argument pour faire une clé sur 32 bits :

31…24	23…16	15…0
Numéro projet & `0xFF`	Mineur périphérique & `0xFF`	Numéro i-nœud & `0xFFFF`

La fonction `ftok()` ne garantit pas réellement l'unicité de la clé, car plusieurs liens matériels sur le même fichier renvoient le même numéro d'i-nœud. De plus, la restriction au numéro

mineur de périphérique ainsi que l'utilisation seulement des 16 bits de poids faibles de l'i-nœud rendent possible – quoique très improbable – l'existence de fichiers différents renvoyant la même clé.

Une fois qu'on a obtenu une clé ou qu'on a choisi d'utiliser une ressource privée avec `IPC_PRIVATE`, on demande l'accès à l'IPC proprement dite.

Ouverture de l'IPC

L'obtention de la ressource IPC se fait à l'aide de l'une des trois commandes `msgget()`, `shmget()` et `semget()`. Les détails d'appel seront précisés plus bas, mais ces fonctions demandent au système de créer éventuellement la ressource si elle n'existe pas, puis de renvoyer un numéro d'identification. Si la ressource existe déjà et si le processus appelant n'a pas les autorisations nécessaires pour y accéder, les routines échouent en renvoyant –1.

À partir de l'identifiant ainsi obtenu, il sera possible respectivement :

- d'envoyer et de recevoir des messages dans une file, à l'aide des fonctions `msgsnd()`, et `msgget()` ;
- d'attacher puis de détacher un segment de mémoire partagée dans l'espace d'adressage du processus avec `shmat()` ou `shmdt()` ;
- de lever de manière bloquante ou non un sémaphore, puis de le relâcher avec la fonction commune `semop()`.

Attention

Il faut bien comprendre que l'emploi de `IPC_PRIVATE` dans `msgget()`, `shmget()` ou `semget()` n'empêche pas l'accès à la ressource par un autre processus, mais garantit uniquement qu'une nouvelle ressource sera créée. En effet, l'identifiant renvoyé par la routine d'ouverture n'aura rien d'exceptionnel et un autre processus pourra très bien l'employer – à condition bien sûr d'avoir les autorisations d'accès nécessaires.

Contrôle et paramétrage

Les IPC proposent quelques options de paramétrage spécifiques au type de communication, ou générales. Pour cela, il existe trois fonctions, `msgctl()`, `shmctl()` et `semctl()`, qui permettent de consulter des attributs regroupés dans des structures `msqid_ds`[1], `shmid_ds` et `semid_ds`.

Dans tous les cas, ces structures permettent l'accès à un objet de type **`struct ipc_perm`**, défini ainsi dans `<sys/ipc.h>` :

Nom	Type	Signification
`__key`	`key_t`	Clé associée à la ressource IPC
`__seq`	`unsigned short`	Numéro de séquence, utilisé de manière interne par le système, à ne pas toucher
`mode`	`unsigned short`	Autorisations d'accès à la ressource, comme pour les permissions des fichiers
`uid`	`uid_t`	UID effectif de la ressource IPC
`gid`	`gid_t`	GID effectif de la ressource IPC

1. Attention il s'agit bien de `msqid_ds` – *message queue identifier data structure* – et non de `msgid_ds` comme on pourrait s'y attendre.

Nom	Type	Signification
cuid	uid_t	UID du créateur de la ressource
cgid	gid_t	GID du créateur de la ressource

Ces informations permettent bien entendu de contrôler l'accès à la ressource IPC. Les modifications de mode ne peuvent être réalisées que par le propriétaire, le créateur de la ressource ou par un processus ayant la capacité **CAP_IPC_OWNER**.

Les fonctions de contrôle permettent également de détruire une ressource IPC. En effet, une file de messages, un ensemble de sémaphores ou une zone de mémoire partagée restent présents dans le noyau même s'il n'y a plus de processus qui les utilisent. Ceci présente l'avantage d'une persistance des données entre deux lancements de la même application – et peut par ailleurs être utilisé par des processus administratifs comme des démons – mais pose aussi l'inconvénient d'une utilisation croissante de la mémoire du noyau sans libération automatique. Il est donc possible de demander explicitement la destruction d'une ressource IPC. Les processus en train de l'employer recevront une indication d'erreur lors de la tentative d'accès suivante.

Files de messages

Les files de messages sont des listes chaînées gérées par le noyau pour contenir des données organisées sous forme d'un type suivi d'un bloc de message proprement dit. Cette représentation complique un peu la manipulation des messages, mais permet – grâce au type transmis – de les hiérarchiser par priorité ou d'obtenir un multiplexage en distinguant plusieurs processus destinataires différents lisant la même file de messages.

Le noyau gère un maximum de **MSGMNI** files indépendantes chacune pouvant comporter des messages de tailles inférieures à **MSGMAX**, soit 8 192 octets. Pour accéder à une file existante ou en créer une nouvelle, on appelle **msgget()**, déclarée ainsi dans `<sys/msg.h>` :

```
int msgget (key_t key, int attribut);
```

Cette routine renvoie l'identificateur de la file de messages demandée, ou –1 en cas d'erreur. Le premier argument doit comprendre une clé caractérisant la file désirée, construite en général à l'aide de la fonction `ftok()` que nous avons examinée plus haut. Si on veut être sûr de créer une nouvelle file qui ne sera pas utilisée en dehors du processus en cours et de ses descendants s'il appelle `fork()`, on peut passer la valeur `IPC_PRIVATE`.

Le second argument peut être vu comme un équivalent grossier des deux derniers arguments de l'appel-système `open()`. Il s'agit d'une composition binaire des constantes suivantes :

Nom	Signification
IPC_CREAT	Créer une nouvelle file s'il n'y en a aucune présentement associée à la clé transmise en premier argument.
IPC_EXCL	Toujours créer une nouvelle file. La fonction `msgget()` échouera si une file existe déjà avec la clé indiquée.

Dans cet argument s'insèrent également les permissions d'accès à la file créée, avec le même format que le dernier argument de `open()`. Seuls les 9 bits de poids faibles sont pris en compte

(pas de Set-UID, de Set-GID, ni de Sticky bit), et les autorisations d'exécution n'ont pas de signification.

> **Attention**
> N'oubliez pas d'introduire ces autorisations d'accès lors de la création, sinon la file aura les permissions 000, ce qui la rend pour le moins difficile à utiliser !

On emploiera donc les combinaisons suivantes :

- Pour avoir une file uniquement réservée au processus appelant et à ses descendants :

```
file = msgget (IPC_PRIVATE, 0x600);
```

  Si le processus est installé Set-UID ou Set-GID et si lui ou ses descendants comptent changer d'identité ultérieurement, on modifiera le mode d'accès ou l'appartenance en conséquence.

- Pour accéder à une file permettant de dialoguer avec d'autres processus du même ensemble d'applications :

```
ma_cle = (key_t) NOMBRE_MAGIQUE;
file = msgget(ma_cle, IPC_CREAT | 0x660);
```

  Les autres processus utiliseront le même nombre magique pour accéder à la file. Il faudra se prémunir contre les risques de conflits, en documentant le nombre employé et en laissant l'utilisateur le modifier, par exemple à l'aide d'une option en ligne de commande.

- Pour s'assurer de la création d'une nouvelle file, cas d'un processus serveur ou d'un démon par exemple :

```
ma_cle = ftok(argv[0], 0);
file = msgget(ma_cle, IPC_CREAT | IPC_EXCL | 0x622);
```

  Ici, on autorise tous les utilisateurs du système à nous envoyer des messages, mais seul le créateur de la file peut les lire. La file est identifiée par une clé construite autour du nom de l'application principale.

  Symétriquement, un processus qui doit uniquement utiliser une file existante, si le processus serveur l'a déjà créée, emploiera :

```
ma_cle = ftok(fichier_executable_serveur, 0);
file = msgget(ma_cle, 0);
```

Naturellement, on vérifie ensuite les cas d'erreur, en surveillant si le retour de `msgget()` est négatif et, si c'est le cas, en examinant `errno` :

Valeur dans `errno`	Signification
`ENOMEM`	Pas assez de mémoire pour allouer les structures de contrôle dans le noyau.
`ENOSPC`	On veut créer une nouvelle file, mais la limite `MSGMNI` est atteinte.
`ENOENT`	La file demandée n'existe pas, et on n'a pas précisé l'attribut `IPC_CREAT`.
`EEXIST`	La file existe et on a demandé un accès exclusif avec l'attribut `IPC_EXCL`.
`EACCES`	La file existe, mais ses autorisations ne permettent pas d'y accéder.
`EIDRM`	La file existe, mais elle est en cours de suppression.

Une fois que la file est créée, on peut – sous réserve d'avoir les autorisations adéquates – y écrire des messages. Un message est une zone de mémoire contiguë contenant un entier `long` représentant le type du message, suivi des données proprement dites. Le type du message, qui doit être supérieur à zéro, est simplement une description interne à l'application, qui n'a pas de signification pour le noyau. On l'emploiera pour filtrer les messages à l'arrivée.

La méthode la plus simple est donc de définir une structure regroupant le type du message et les données qu'on veut envoyer :

```
typedef struct {

    /* Type pour msgsnd() et msgrcv()  */
    long   type;

    /* Données de l'application        */
    char   identifiant[25];
    double x;
    double y;
    double vitesse;
    time_t estimation_arrivee;

} mon_message_t;
```

La structure envoyée ne peut naturellement pas comprendre de pointeurs puisqu'ils n'auraient aucune signification dans l'espace d'adressage du processus récepteur. La transmission d'une chaîne de caractères doit autant que possible se faire en employant une zone allouée automatiquement dans la structure, comme le champ `identifiant` de notre exemple ci-dessus. La transmission de données allouées dynamiquement est assez compliquée puisqu'elle nécessite la réservation d'un espace de la dimension d'un `long int` avant la zone véritablement utilisée.

L'envoi d'un message se fait par l'intermédiaire de l'appel-système **`msgsnd()`**, déclaré ainsi :

```
int msgsnd (int file, const void * message, int taille, int attributs);
```

Le numéro de file indiqué doit avoir été fourni préalablement par `msgget()`, le second argument pointe sur le message tel que nous venons de le décrire, et le troisième argument indique la longueur utile du message *sans compter son type*. En dernière position, le seul attribut qu'on puisse transmettre éventuellement est **`IPC_NOWAIT`**, pour que l'appel-système ne soit pas bloquant. Sinon, s'il n'y a pas assez de place dans la file pour stocker le message, `msgsnd()` reste en attente.

La valeur de retour de `msgsnd()` est zéro si tout s'est bien passé, et –1 sinon. Dans ce cas `errno` peut contenir l'un des codes suivants :

Valeur dans errno	Signification
`EACCES`	Le processus doit avoir l'autorisation d'écrire dans la file, et ce n'est pas le cas.
`EAGAIN`	On a demandé une émission non bloquante avec `IPC_NOWAIT`, et il n'y a pas assez de place dans la file pour le moment.
`EIDRM`	La file a été supprimée.
`EINTR`	Un signal a interrompu l'appel-système avant qu'il n'ait pu écrire quoi que ce soit.
`EFAULT, EINVAL`	Argument de `msgsnd()` ou type de message invalide.
`ENOMEM`	Manque de mémoire dans le noyau pour stocker le message.

Le message est copié dans la file, aussi il est possible d'écraser les données originales dès le retour de `msgsnd()`.

La lecture dans une file de messages se fait en invoquant l'appel-système **`msgrcv()`**, déclaré ainsi :

```
int msgrcv (int file, void * message, int taille,
            long type, int attributs);
```

Les premiers arguments ont la même signification que dans `msgsnd()`, le troisième indiquant la taille maximale de la zone de données du message à lire, qui doit donc être disponible en seconde position. On emploie généralement `MSGMAX`, qui correspond à la taille maximale d'un message sur le système, ou une dimension fixée si tous les messages sont constitués autour de la même structure.

> **Attention**
> La taille ne prend pas en compte le type du message, il s'agit uniquement de la zone utile du message.

Le type indiqué en quatrième position permet de sélectionner les messages qu'on désire recevoir. Le comportement de `msgrcv()` varie en fonction de cette valeur :

- Un type nul indique qu'on veut recevoir le prochain message disponible dans la file. C'est le comportement habituel d'une file de messages, où on traite les données dans l'ordre d'arrivée.
- Un type positif permet de réclamer le premier message dudit type disponible dans la file. Cette méthode donne la possibilité de multiplexer plusieurs processus en écriture et plusieurs en lecture sur la même file. Chacun d'eux utilise un identifiant unique – par exemple son PID –, et la file permet à n'importe quel processus d'envoyer un message vers un destinataire précis.
- Un type négatif sert à réclamer le premier message disponible ayant le plus petit type inférieur ou égal à la valeur absolue de ce quatrième argument. Il est ainsi possible d'introduire des priorités entre les messages. Le message avec le plus faible type (1) sera délivré avant tous les autres, même s'ils sont en attente depuis plus longtemps.

Le dernier argument peut contenir un OU binaire avec les constantes suivantes :

Nom	Signification
`IPC_NOWAIT`	Ne pas rester en attente si aucun message du type réclamé n'est disponible, mais au contraire échouer avec l'erreur `ENOMSG`.
`MSG_EXCEPT`	Réclame un message de n'importe quel type *sauf* celui qui est indiqué en quatrième argument, qui doit être nécessairement strictement positif.
`MSG_NOERROR`	Si le message extrait est trop long, il sera tronqué sans que l'erreur `E2BIG` se produise, contrairement au comportement par défaut.

Lorsqu'elle réussit cette fonction renvoie le nombre d'octets du message – non compris son type – et –1 lorsqu'elle échoue. Dans ce cas, `errno` peut contenir les codes `EFAULT`, `EIDRM`, `EINTR` ou `EINVAL`, avec les mêmes significations que pour `msgsnd()`, ou :

- `EACCES` : le processus n'a pas la permission de lecture sur la file.
- `ENOMSG` : aucun message disponible lors d'une lecture avec l'attribut `IPC_NOWAIT`.

- `E2BIG` : le message disponible est trop grand pour tenir dans la zone transmise.

Finalement, le contrôle et le paramétrage d'une file de messages se font à l'aide de la fonction **`msgctl()`**. Celle-ci est déclarée ainsi :

```
int msgctl (int file, int commande, struct msqid_ds * attributs);
```

Il y a trois commandes possibles, qu'on passe en second argument :

- **`IPC_STAT`** : pour obtenir les paramètres concernant la file de messages et les stocker dans la structure `msqid_ds` passée en dernière position. Cette structure sera détaillée plus bas. Il faut avoir l'autorisation de lecture sur la file de messages.
- **`IPC_SET`** : pour configurer certains paramètres en utilisant la structure passée en troisième argument. Les paramètres qui sont mis à jour seront décrits ci-dessous. Pour pouvoir modifier ces éléments, il faut que le processus appelant soit le propriétaire ou le créateur de la file de messages, ou qu'il ait la capacité `CAP_SYS_ADMIN`.
- **`IPC_RMID`** : pour supprimer la file de messages. Tous les processus en attente de lecture ou d'écriture sur la file seront réveillés. Les opérations ultérieures d'accès à cette file échoueront. Il y a toutefois un risque qu'une nouvelle file soit créée par la suite et que le noyau lui attribue le même identifiant. Si un processus attend longtemps avant d'accéder à la file supprimée, il risque de se trouver en face de la nouvelle file sans s'y attendre. Ce manque de fiabilité est l'un des arguments employés par les détracteurs des IPC Système V.

La structure `msqid_ds` est définie dans `<sys/msg.h>`. Ses membres susceptibles de nous intéresser sont :

Nom	Type	Signification
`msg_perm`	`struct ipc_perm`	Autorisations d'accès à la file de messages
`msg_stime`	`time_t`	Heure du dernier `msgsnd()` sur la file
`msg_rtime`	`time_t`	Heure du dernier `msgrcv()` sur la file
`msg_ctime`	`time_t`	Heure du dernier paramétrage de la file
`msg_qnum`	`unsigned short`	Nombre de messages actuellement présents dans la file
`msg_qbytes`	`unsigned short`	Taille maximale en octets du contenu de la file
`msg_lspid`	`Pid_t`	PID du processus ayant effectué le dernier `msgsnd()`
`msg_lrpid`	`Pid_t`	PID du processus ayant effectué le dernier `msgrcv()`

Rappelons que le détail de la structure `ipc_perm` composant le premier membre a été développé dans la section précédente. Il existe d'autres champs dans la structure `msqid_ds`, mais ils sont plutôt réservés à l'usage interne du noyau.

Lorsqu'on utilise la commande `IPC_SET`, les membres suivants sont mis à jour : `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` et `msg_qbytes`. Si toutefois cette dernière valeur est supérieure à la constante **`MSGMNB`** (16 Ko par défaut), le processus doit nécessairement avoir la capacité `CAP_SYS_RESOURCE`.

Nous allons construire trois petits programmes servant d'interface en ligne de commande pour `msgsnd()`, `msgrcv()` et `msgctl()` avec la commande `IPC_RMID`. Le premier prend en arguments le nom d'un fichier servant pour créer la clé IPC, une valeur indiquant le type du message, et une chaîne de caractères composant le corps du message émis.

exemple_msgsnd.c :

```
    #include <stdio.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    typedef struct {
        long type;
        char texte [256];
    } message_t;

    int
main (int argc, char * argv [])
{
    key_t     key;
    message_t message;
    int       file;

    if (argc != 4) {
        fprintf(stderr, "Syntaxe : %s fichier_clé type message \n",
                            argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((key = ftok(argv[1], 0)) == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    if ((sscanf(argv [2], "%ld", & (message.type)) != 1)
     || (message.type <= 0)) {
        fprintf(stderr, "Type invalide");
        exit(EXIT_FAILURE);
    }
    strncpy (message.texte, argv[3], 255);
    message.texte[255] = '\0';
    if ((file = msgget(key, IPC_CREAT | 0600)) == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    if (msgsnd(file, (void *) & message, 256, 0) <0) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Le second programme permet de bloquer en attente d'un message. Ses arguments sont le nom du fichier servant de clé et le type du message attendu.

exemple_msgrcv.c :

```
    #include <stdio.h>
    #include <string.h>
```

```
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    typedef struct {
        long type;
        char texte [256];
    } message_t;

    int
main (int argc, char * argv[])
{
    key_t     key;
    message_t message;
    int       file;
    long      type;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s fichier_clé type \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((key = ftok(argv[1], 0)) == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    if (sscanf(argv[2], "%ld", & type) != 1) {
        fprintf(stderr, "Type invalide");
        exit(EXIT_FAILURE);
    }
    if ((file = msgget(key, IPC_CREAT | 0600)) == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    if (msgrcv(file, (void *) & message, 256, type, 0) >= 0)
        fprintf(stdout, "(%ld) %s \n", message.type, message.texte);
    else
        perror("msgrcv");
    return EXIT_SUCCESS;
}
```

Finalement notre troisième programme servira uniquement à détruire une file, en prenant en argument le nom du fichier servant de clé.

exemple_msgctl.c :

```
    #include <stdio.h>
    #include <string.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>

    int
main (int argc, char * argv[])
```

```
{
    key_t key;
    int   file;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier_clé \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((key = ftok(argv[1], 0)) == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    if ((file = msgget(key, 0)) == -1)
        exit(EXIT_SUCCESS);
    msgctl(file, IPC_RMID, NULL);
    return EXIT_SUCCESS;
}
```

Voyons à présent l'utilisation de ces commandes. Tout d'abord, nous allons simplement envoyer quelques messages et les récupérer dans l'ordre d'émission. Nous utilisons le fichier exécutable `exemple_msgsnd` pour créer la clé d'accès à la file :

```
$ ./exemple_msgsnd ./exemple_msgsnd 1 "Message 1"
$ ./exemple_msgsnd ./exemple_msgsnd 2 "Message 2"
$ ./exemple_msgsnd ./exemple_msgsnd 1 "Message 3"
$ ./exemple_msgrcv ./exemple_msgsnd 0
(1) Message 1
$ ./exemple_msgrcv ./exemple_msgsnd 0
(2) Message 2
$ ./exemple_msgrcv ./exemple_msgsnd 0
(1) Message 3
$
```

Avec un argument `type` nul lors de la lecture, les messages sont simplement extraits dans l'ordre d'arrivée, sans tenir compte de leur type réel. À présent, nous allons utiliser un `type` positif lors de la lecture, afin de vérifier qu'on peut ainsi filtrer le destinataire. La lecture étant bloquante, nous l'interrompons par un Contrôle-C lorsqu'il n'y aura plus de messages disponibles.

```
$ ./exemple_msgsnd ./exemple_msgsnd 1 "1er message pour 1"
$ ./exemple_msgsnd ./exemple_msgsnd 1 "2ème message pour 1"
$ ./exemple_msgsnd ./exemple_msgsnd 2 "1er message pour 2"
$ ./exemple_msgsnd ./exemple_msgsnd 2 "2ème message pour 2"
$ ./exemple_msgsnd ./exemple_msgsnd 2 "3ème message pour 2"
$ ./exemple_msgrcv ./exemple_msgsnd 1
(1) 1er message pour 1
$ ./exemple_msgrcv ./exemple_msgsnd 1
(1) 2ème message pour 1
$ ./exemple_msgrcv ./exemple_msgsnd 1
  (Contrôle-C)
$ ./exemple_msgrcv ./exemple_msgsnd 2
(2) 1er message pour 2
```

```
$ ./exemple_msgrcv ./exemple_msgsnd 2
(2) 2ème message pour 2
$ ./exemple_msgrcv ./exemple_msgsnd 2
(2) 3ème message pour 2
$ ./exemple_msgrcv ./exemple_msgsnd 2
  (Contrôle-C)
$
```

Nous voyons bien qu'une lecture réclamant un type 1 restera bloquée s'il n'y a pas de message de ce type disponible, même s'il y en a de type 2. Il est possible ainsi d'implémenter un serveur lisant des requêtes provenant de multiples clients et leur répondant dans la même file de messages. Chaque client insère son propre PID dans le message requête et l'envoie avec le type 1. Le serveur lit les messages de type 1 et y répond en leur attribuant comme type le PID du client concerné. Chacun peut ainsi filtrer les messages qui le concernent. Naturellement, le PID numéro 1 étant réservé au processus *init*, il n'y a pas d'ambiguïté lors des interrogations.

Maintenant nous allons illustrer la possibilité de gérer des priorités en transmettant un argument `type` négatif lors de la lecture.

```
$ ./exemple_msgsnd ./exemple_msgsnd 3 "1er Message"
$ ./exemple_msgsnd ./exemple_msgsnd 2 "2ème Message"
$ ./exemple_msgsnd ./exemple_msgsnd 5 "3ème Message"
$ ./exemple_msgsnd ./exemple_msgsnd 2 "4ème Message"
$ ./exemple_msgsnd ./exemple_msgsnd 1 "5ème Message"
$ ./exemple_msgrcv ./exemple_msgsnd -4
 (1) 5ème Message
$ ./exemple_msgrcv ./exemple_msgsnd -4
(2) 2ème Message
$ ./exemple_msgrcv ./exemple_msgsnd -4
(2) 4ème Message
$ ./exemple_msgrcv ./exemple_msgsnd -4
(3) 1er Message
$ ./exemple_msgrcv ./exemple_msgsnd -4
  (Contrôle-C)
$ ./exemple_msgrcv ./exemple_msgsnd -5
(5) 3ème Message
$
```

Nous voyons que les messages sont bien extraits dans l'ordre des priorités croissantes ou dans l'ordre d'arrivée en cas d'égalité (type 2 dans notre exemple). On remarque qu'en réclamant un type inférieur ou égal à 4 lors de la lecture, on n'obtient jamais les messages de priorité supérieure (5 en l'occurrence).

Enfin, nous allons examiner ce qui se passe lors de la suppression d'une file alors qu'un processus est en attente de lecture. On utilise un second terminal représenté ici en deuxième colonne pour invoquer `exemple_msgctl`.

```
$ ./exemple_msgrcv exemple_msgsnd 0
                                        $ ./exemple_msgctl exemple_msgsnd
msgrcv: Identificateur éliminé          $
$
```

Le message affiché par `perror()` correspond à l'erreur `EIDRM`.

Étant donné que les files d'attente, comme les autres ressources IPC par ailleurs, sont persistantes jusqu'à leur suppression par `msgctl()` ou par l'arrêt du système, il existe deux utilitaires d'administration, `/usr/bin/ipcs` et `/usr/bin/ipcrm`, permettant de les manipuler. La première commande affiche la liste de toutes les ressources IPC en cours d'utilisation sur le système, avec des informations sur le propriétaire, l'espace occupé, etc. Le second utilitaire permet de supprimer une ressource – à condition d'en avoir le droit – en indiquant le type de ressource et son identifiant, fourni par `ipcs`. En voici un exemple :

```
$ ./exemple_msgsnd exemple_msgsnd 1 "Message"
$ ipcs
------ Shared Memory Segments --------
key       shmid     owner     perms     bytes     nattch    status

------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status

------ Message Queues --------
key       msqid     owner     perms     used-bytes  messages
0x0005e931 1280      ccb       600       256         1

$ ipcrm
usage: ipcrm [shm | msg | sem] id
$ ipcrm msg 1280
resource deleted
$ ipcs
------ Shared Memory Segments --------
key       shmid     owner     perms     bytes     nattch    status

------ Semaphore Arrays --------
key       semid     owner     perms     nsems     status

------ Message Queues --------
key       msqid     owner     perms     used-bytes  messages

$
```

On peut se demander comment l'utilitaire `ipcs` arrive à obtenir la liste des IPC présentes.

Cette commande balaye en fait les identifiants compris entre 0 et le nombre maximal de files autorisées sur le système, et invoque `msgctl()` avec la commande `MSG_STAT` sur chacun d'eux. Cette commande échoue si l'identifiant n'est pas utilisé.

Pour obtenir des informations globales, telles que le nombre maximal de messages par file, l'utilitaire s'appuie sur une structure d'information particulière nommée `msginfo`, qu'on emploie à la place de la structure `msqid_ds` dans l'appel `msgctl()`, avec une conversion explicite de type. La commande utilisée est alors `MSG_INFO`, qui demande au noyau de remplir la structure `msginfo`. Celle-ci contient des données diverses sur les limites imposées aux files de messages, ainsi que sur l'état actuel de leur utilisation. Il existe des équivalents de cette structure d'information pour les autres ressources IPC. Ce mécanisme est propre à Linux.

Le fait que les files de messages continuent à occuper de la mémoire dans le noyau tant qu'elles n'ont pas été explicitement détruites est un inconvénient important. Un autre gros défaut de ce moyen de communication est l'impossibilité de faire dialoguer des applications

se trouvant sur des machines différentes. Nous verrons que la solution BSD au problème de communication entre processus (le système des *sockets*) est nettement plus avantageuse car elle permet de manière transparente de faire dialoguer des processus répartis sur différents systèmes. Il existe toutefois un domaine où les IPC Système V sont encore largement employés : le partage de segment de mémoire.

Mémoire partagée

Nous avons déjà observé dans le chapitre 14 une méthode permettant de partager une zone de mémoire entre deux processus distincts, en employant `mmap()` avec l'attribut `MAP_SHARED` pour projeter un fichier de la dimension voulue. L'inconvénient de cette technique est d'une part que la mémoire n'est partagée qu'entre processus issus d'un même père. L'appel `mmap()` doit être réalisé avant la séparation des processus par `fork()`. D'autre part, cette projection en mémoire n'étant pas conservée lors d'un `exec()`, il n'est pas possible de partager des données entre des programmes exécutables différents.

Rappelons également que la solution extrême au partage de mémoire entre portions de code différentes – les threads – n'est pas non plus applicable pour exécuter des programmes distincts.

En fait, le système de la mémoire partagée offert par les IPC Système V est assez élégant :

- Une fonction **`shmget()`** permet à partir d'une clé `key_t` d'obtenir l'identifiant d'un segment de mémoire partagée existant ou d'en créer un au besoin.
- L'appel-système **`shmat()`** permet d'attacher le segment dans l'espace d'adressage du processus.
- La fonction **`shmdt()`** sert à détacher le segment si on ne l'utilise plus.
- Enfin, l'appel-système **`shmctl()`** permet de paramétrer ou de supprimer un segment partagé.

Les prototypes de ces routines sont déclarés dans `<sys/shm.h>` ainsi :

```
int    shmget (key_t key, int taille, int attributs);
char * shmat  (int identifiant, char * adresse, int attributs);
int    shmdt  (char * adresse);
int    shmctl (int identifiant, int commande,
                struct shmid_ds * attributs);
```

L'appel-système `shmget()` fonctionne globalement comme `msgget()`, en employant la clé transmise en premier argument pour rechercher ou créer un bloc de mémoire partagée. Les attributs indiqués en dernière position comportent les 9 bits de poids faibles de l'autorisation d'accès, et éventuellement les constantes `IPC_CREAT` et `IPC_EXCL` qui ont les mêmes significations qu'avec les files de messages. Les erreurs renvoyées par cette fonction sont équivalentes à celles de `msgget()`.

Le second argument de cette routine est la taille du segment désiré, en octets. Cette taille sert lors de la création d'une nouvelle zone de mémoire partagée. La valeur indiquée est arrondie au multiple supérieur de la taille des pages mémoire sur le système (4 Ko sur un PC). Si la taille demandée lors de la création est inférieure à la valeur **`SHMMIN`** ou supérieure à **`SHMMAX`**, une erreur se produit. Pour accéder à une zone mémoire déjà existante, il faut demander une valeur inférieure ou égale à la taille effective du segment. On emploie généralement zéro dans ce cas, car le système ne réduit pas la taille de la projection d'un segment existant.

Une fois obtenu l'identifiant d'un segment partagé, on doit l'attacher dans l'espace mémoire du processus à l'aide de la fonction `shmat()`. On indique en second argument l'adresse désirée pour l'attachement. Si cette adresse est nulle, le noyau recherche un emplacement libre dans l'espace d'adressage du processus, y réalise la projection, et l'appel-système `shmat()` renvoie l'adresse du premier octet de la zone partagée. C'est bien entendu le mécanisme qu'on utilisera toujours. Le fait de mentionner explicitement une adresse d'attachement ne se justifie que dans des cas exceptionnels (par exemple des émulateurs ou des débogueurs) ne nous concernant pas ici. Mentionnons quand même que l'adresse transmise dans ce cas doit être alignée sur une frontière de page, ou alors il faut passer l'attribut **`SHM_RND`** dans le dernier argument pour demander au noyau d'arrondir l'adresse indiquée à la limite de page inférieure.

L'attachement peut être réalisé en lecture seule si l'attribut **`SHM_RDONLY`** est passé en troisième argument de `shmat()`, sinon la projection est réalisée en lecture et écriture.

La fonction `shmctl()` permet, à la manière de `msgctl()`, d'agir sur un segment partagé. La commande employée en seconde position peut être :

- **`IPC_STAT`** : pour remplir la structure `shmid_ds` que nous allons détailler ci-dessous.
- **`IPC_SET`** : pour modifier l'appartenance ou les autorisations d'accès au segment.
- **`IPC_RMID`** : pour supprimer le segment. Ce dernier est alors marqué comme « prêt pour la suppression », mais ne sera effectivement détruit qu'une fois qu'il aura été détaché par le dernier processus qui l'utilise. Cela signifie aussi que tant qu'un processus conserve le segment attaché, il est toujours possible de le lier à nouveau avec `shmat()`, même s'il a été marqué pour la destruction.
- **`SHM_LOCK`** : permet de verrouiller le segment en mémoire pour s'assurer qu'il ne sera pas envoyé sur le périphérique de swap. Nous avons déjà étudié ce mécanisme dans le chapitre 14 avec l'appel-système `mlock()`. Cette opération réduisant la mémoire vive disponible pour les autres processus, elle est privilégiée et nécessite un UID nul ou la capacité `CAP_IPC_LOCK`.
- **`SHM_UNLOCK`** : permet symétriquement de déverrouiller une page de la mémoire, autorisant à nouveau son transfert en mémoire secondaire.

La structure `shmid_ds` contenant les paramètres associés au segment de mémoire partagée comprend notamment les membres suivants :

Nom	Type	Signification
`shm_perm`	`struct ipc_perm`	Autorisation d'accès au segment de mémoire
`shm_segsz`	`size_t`	Taille en octets du segment
`shm_atime`	`time_t`	Heure du dernier attachement
`shm_dtime`	`time_t`	Heure du dernier détachement
`shm_ctime`	`time_t`	Heure de la dernière modification des autorisations
`shm_cpid`	`pid_t`	PID du processus créateur du segment
`shm_lpid`	`pid_t`	PID du processus ayant réalisé la dernière intervention
`shm_nattch`	`unsigned short`	Nombre actuel d'attachements en mémoire

On remarquera que l'utilisation des segments de mémoire partagée est le mécanisme de communication entre processus le plus rapide, car il n'y a pas de copie des données transmises. On évite notamment le transfert des informations entre l'espace mémoire de l'utilisateur et l'espace mémoire du noyau, à la différence de `msgsnd()` sur les files de messages, ou même de `write()` lors de l'utilisation de sockets BSD. Ce procédé de communication est donc parfaitement adapté au partage de gros volumes de données entre processus distincts.

Nous avons signalé que la taille demandée lors de la création d'un segment est arrondie au multiple supérieur de la dimension d'une page. Cette dimension représente 4 Ko sur un PC, ce qui n'est pas négligeable. Ceci implique qu'il faut éviter de créer trop de petits segments partagés, pour éviter de gâcher des ressources mémoire. Si on veut partager simultanément de multiples variables, on les regroupera au sein d'un tableau ou d'une structure pour les réunir sur une même page mémoire.

Comme nous l'avons vu lors de notre étude des threads dans le chapitre 12, il est indispensable, lors de l'accès à des ressources communes, de synchroniser les différents acteurs, pour éviter les interférences regrettables. Pour cela, on dispose d'un dernier mécanisme IPC servant à organiser l'utilisation des mémoires partagées : les sémaphores.

Nous attendrons donc la fin de la prochaine section pour présenter un exemple d'utilisation des appels-système examinés précédemment.

Sémaphores

Nous avons déjà rencontré, dans le chapitre 12, les sémaphores temps-réel, permettant la synchronisation de différents threads. Le principe reste quasiment le même ici, mais les fonctions sont totalement différentes.

Un sémaphore est dans sa forme la plus simple un drapeau qui peut être levé ou baissé. Il sert à contrôler l'accès à une ressource critique grâce à deux opérations :

- Avant l'accès, un processus attend que le drapeau soit levé, puis il le baisse.
- Après avoir utilisé la ressource protégée, le processus relève le drapeau, et le noyau réveille les autres processus bloqués dans l'opération précédente.

Le test qui intervient dans la première de ces opérations est atomiquement lié à la modification qui le suit. Ceci garantit qu'en aucun cas deux processus ne verront simultanément le drapeau baissé et se l'attribueront.

Ces deux opérations sont généralement notées *P()* et *V()* en abréviation des traductions des termes *tester* et *incrémenter* en hollandais, langue natale de l'inventeur des sémaphores, Edsger Dijkstra. Dans l'implémentation Système V des sémaphores, elles sont toutefois quelque peu compliquées :

Un sémaphore peut servir à contrôler non plus une simple ressource critique mais un accès à plusieurs exemplaires de la même ressource. Ainsi le drapeau est remplacé par un compteur, qu'on augmente ou diminue d'une valeur entière qui n'est pas nécessairement 1. L'opération *Pn()* est alors bloquante tant que le compteur du sémaphore est inférieur à la valeur *n* demandée, puis elle diminue le compteur de cette quantité. Parallèlement, l'opération *Vn()* doit incrémenter le compteur de la valeur *n* d'exemplaires libérés de la ressource.

Même lorsque chaque opération *P()* ou *V()* est invoquée avec $n = 1$, l'utilisation d'un compteur associé au sémaphore est intéressante, puisque ce mécanisme permet d'autoriser l'accès

simultané d'un nombre limité de processus à une quantité donnée de ressources. Par exemple, on peut limiter la consommation CPU d'un logiciel de calculs parallèles intensifs sur une machine multiprocesseur, en décidant de ne laisser simultanément que trois processus ou threads concurrents en exécution. On positionne le compteur initial à 3, et chaque fil d'exécution appellera *P()*, puis *V()* avec une seule unité.

Les opérations offertes par les IPC Système V permettent de manipuler des ensembles de sémaphores. Il est possible de demander en une fois des opérations *Pn()* ou *Vn()* indépendantes sur chaque sémaphore d'un ensemble. Ces opérations sont liées atomiquement, ce qui signifie que le noyau les réalisera toutes ou n'en exécutera aucune. Il peut aussi rester bloqué longuement en attente d'une ressource.

Finalement, étant donné qu'un processus peut se terminer à tout moment – notamment à cause d'un signal –, il est important de relâcher automatiquement les sémaphores qu'il maintenait. Pour cela, les IPC proposent un mécanisme d'annulation programmable pour chaque action. Au moment de la fin du processus, le noyau effectue l'opération inverse de celle qui a été réalisée. Mais ceci complique à nouveau l'utilisation des sémaphores.

Les routines servant à manipuler les sémaphores sont **`semget()`**, qui accomplit une tâche comparable à `msgget()` ou à `shmget()`, **`semop()`**, qui regroupe les opérations *Pn()* et *Vn()*, et **`semctl()`**, qui permet entre autres de configurer ou de supprimer un ensemble de sémaphores. Leurs prototypes sont déclarés dans `<sys/sem.h>` ainsi :

```
int semget (key_t key, int nombre, int attributs);
int semop (int identifiant, struct sembuf * operation, unsigned nombre);
int semctl (int identifiant, int numero,
            int commande, union semun attributs);
```

L'appel-système `semget()` fonctionne comme ses confrères `msgget()` et `shmget()`, avec simplement en second argument le nombre de sémaphores dans l'ensemble. Cette valeur n'est prise en compte que lors de la création de la ressource, pas au moment de l'accès à un ensemble existant. Le troisième argument peut contenir comme d'habitude `IPC_CREAT`, `IPC_EXCL` et les autorisations d'accès.

La routine `semop()` sert à la fois pour les opérations *Pn()* et *Vn()* sur de multiples sémaphores appartenant au jeu indiqué en premier argument. Chaque opération est décrite par une structure **`sembuf`**, définie dans `<sys/sem.h>` ainsi :

Nom	Type	Signification
`sem_num`	`short int`	Numéro du sémaphore concerné dans l'ensemble. La numérotation débute à zéro.
`sem_op`	`short int`	Valeur numérique correspondant à l'opération à réaliser.
`sem_flg`	`short int`	Attributs pour l'opération.

L'opération effectuée est déterminée ainsi :

- Lorsque le champ `sem_op` d'une structure `sembuf` est strictement positif, le noyau incrémente le compteur interne associé au sémaphore de la valeur indiquée et réveille les processus en attente.

  Quand `sembuf.sem_op` = *n*, avec *n* > 0, alors l'opération est *Vn()*.

- Lorsque le champ `sem_op` est strictement négatif, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit supérieur à `sem_op`, puis il décrémente le compteur de cette valeur avant de continuer l'exécution du processus.

  Quand `sembuf.sem_op` = *n*, avec *n* < 0, alors l'opération est *Pn()*.
- Lorsque le champ `sem_op` est nul, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit nul, puis il continue l'exécution du programme. Cette fonctionnalité permet de synchroniser les processus.

Il existe deux options possibles pour le membre `sem_flg` :

- **`IPC_NOWAIT`** : l'opération ne sera pas bloquante, même si le champ `sem_op` est négatif ou nul, mais l'appel-système indiquera l'erreur `EAGAIN` dans `errno` si l'opération n'est pas réalisable.
- **`SEM_UNDO`** : pour être sûr que le sémaphore retrouvera un état correct même en cas d'arrêt intempestif du programme, le noyau va mémoriser l'opération inverse de celle qui a été réalisée et l'effectuera automatiquement à la fin du processus. Nous allons préciser ce mécanisme plus loin.

En fait, la routine `semop()` prend en second argument une table de structures `sembuf`. Le nombre d'éléments dans cette table est indiqué en dernière position. Le noyau garantit que les opérations seront atomiquement liées, ce qui signifie qu'elles seront toutes réalisées ou qu'aucune ne le sera. Bien entendu, il suffit qu'une seule opération avec `sem_op` négatif ou nul échoue avec l'attribut `IPC_NOWAIT` pour que toutes les modifications soient annulées.

Pour implémenter les fonctions *P()* et *V()* définies par Dijkstra, on peut donc employer un ensemble avec un seul sémaphore, qu'on manipulera ainsi :

```
int
P (int identifiant)
{
    struct sembuf buffer;
    buffer.sem_num = 0;
    buffer.sem_op  = -1;
    buffer.sem_flg = IPC_UNDO;
    return semop(identifiant, & buffer, 1);
}

    int
V (int identifiant)
{
    struct sembuf buffer;
    buffer.sem_num = 0;
    buffer.sem_op  = 1;
    buffer.sem_flg = IPC_UNDO;
    return semop(identifiant, & buffer, 1);
}
```

L'option `SEM_UNDO` employée lors d'une opération permet au processus de s'assurer qu'en cas de terminaison impromptue alors qu'il bloque un sémaphore le noyau en restituera l'état initial. Ceci est réalisé en utilisant un compteur par sémaphore et par processus qui a demandé un accès à l'ensemble. Ce mécanisme est donc – légèrement – coûteux en mémoire. Le noyau modifie l'état de ce compteur à chaque opération sur le processus en y inscrivant l'opération

inverse. Si par exemple le processus effectue une opération *Pn()*, le noyau le bloque jusqu'à ce que le compteur du sémaphore soit supérieur à *n*, puis il diminue le compteur de cette valeur, et il augmente le compteur d'annulation du sémaphore pour ce processus de la valeur *n*. Lorsque le processus réalisera *Vn()*, le noyau augmentera le compteur du sémaphore et réduira le compteur d'annulation.

Lorsque le processus se termine, le noyau ajoute le compteur d'annulation à celui du sémaphore. Si le processus a bien libéré le sémaphore, le compteur d'annulation est nul, et rien ne se passe. Si par contre le processus s'est terminé après avoir effectué *Pn()*, mais sans avoir réalisé *Vn()*, le compteur d'annulation vaut +*n* et le noyau libère ainsi automatiquement le sémaphore.

Une question peut se poser dans le cas inverse, si le noyau doit décrémenter le compteur du sémaphore lors de la fin d'un processus : doit-il attendre que le compteur du sémaphore soit supérieur à celui d'annulation, au risque de bloquer indéfiniment ? La réponse est loin d'être évidente. L'implémentation actuelle sous Linux consiste à diminuer immédiatement le compteur, mais à limiter ce dernier à zéro. D'autres systèmes peuvent préférer bloquer indéfiniment – à la manière d'un processus zombie qui attend la lecture de son code de retour pour disparaître entièrement – pour garantir l'annulation de n'importe quelle opération.

En réalité, le problème ne devrait pas exister. Hormis quelques cas particuliers servant à des expérimentations sur les sémaphores, on ne devrait jamais invoquer *Vn()* si *Pn()* n'a pas été appelée auparavant. Naturellement, on peut toujours être confronté à des bogues, notamment si on invoque *Pn()*, puis *n* fois *V1()* par exemple.

En fait, dans les applications réelles, l'emploi des sémaphores doit autant que possible être restreint aux opérations *P()* et *V()* sur un seul sémaphore à la fois. On limitera également le compteur du sémaphore à la valeur 1. Avec ces contraintes, l'utilisation des opérations sur les sémaphores ne pose pas de problèmes particuliers. Si le processus risque de bloquer – ou d'être tué – durant la portion critique où il tient un sémaphore, on emploiera l'option `SEM_UNDO`. Bien sûr, si on utilise une fois cette option, on prendra la précaution de le faire à chaque opération sur le sémaphore.

La fonction **`semctl()`** permet de consulter ou de modifier le paramétrage d'un jeu de sémaphore, mais également de fixer l'état du compteur. Cette routine utilise traditionnellement en dernier argument une union définie ainsi :

```
union semun {
    int                 valeur;
    struct semid_ds *   buffer;
    unsigned short int * table;
}
```

En fait, cette union n'est *pas* définie dans les fichiers d'en-tête système, elle doit être déclarée manuellement dans le programme utilisateur. En réalité, le prototype de `semctl()`, vu par le compilateur, est en substance le suivant :

```
int semctl (int identifiant, int numero, int commande, ...);
```

Les points d'élision en fin de liste indiquent la présence éventuelle d'un argument supplémentaire dont le type n'est pas mentionné. On peut donc transmettre n'importe quel type de donnée, c'est la commande indiquée en troisième position qui déterminera la conversion. Pour garder une certaine homogénéité aux appels `semctl()`, on préfère généralement regrouper les diverses possibilités dans une union, qui permet quand même une vérification minimale.

En fonction de la commande, le numéro indiqué en seconde position et l'union en dernier argument auront donc des rôles différents :

Commande	Signification
IPC_STAT	Remplir le membre buffer de l'union semun avec le paramétrage de l'ensemble de sémaphores. Le second argument de semctl() est ignoré.
IPC_SET	Utiliser le membre buffer de l'union semun pour paramétrer les autorisations d'accès sur l'ensemble de sémaphores. Le second argument de semctl() est ignoré.
IPC_RMID	Supprimer l'ensemble de sémaphores. Tous les processus en attente sont réveillés et peuvent recevoir l'erreur EIDRM. Les second et quatrième arguments sont ignorés.
GETALL	Recopier la valeur de tous les sémaphores de l'ensemble dans le membre table de la structure semun. Cette table doit être correctement dimensionnée avec un unsigned short par sémaphore. Le second argument est ignoré.
SETALL	Fixer les compteurs de tous les sémaphores de l'ensemble avec les valeurs contenues dans le membre table de la structure semun. Les processus en attente sur un sémaphore sont réveillés si son compteur augmente. La table doit être correctement dimensionnée avec un unsigned short par sémaphore. Le second argument est ignoré.
GETVAL	Lire la valeur du sémaphore dont le numéro est indiqué dans le second argument de semctl(). Cette valeur est renvoyée, tandis que le quatrième argument est ignoré.
SETVAL	Fixer la valeur du sémaphore dont le numéro est indiqué dans le second argument en employant le contenu du membre valeur de l'union semun. Les processus en attente sont réveillés au besoin.
GETNCNT	Renvoyer le nombre de processus en attente d'augmentation du compteur du sémaphore dont le numéro est indiqué en second argument. Le quatrième argument est ignoré.
GETZCNT	Renvoyer le nombre de processus en attente d'annulation du compteur du sémaphore dont le numéro est indiqué en second argument. Le quatrième argument est ignoré.
GETPID	Renvoyer le PID du processus ayant réalisé la dernière opération sur le sémaphore dont le numéro est indiqué en second argument. Le quatrième argument est ignoré.

La structure **semid_ds** qui représente le paramétrage d'un jeu de sémaphore contient notamment les membres suivants :

Nom	Type	Signification
sem_perm	struct ipc_perm	Autorisations d'accès à l'ensemble de sémaphores
sem_otime	time_t	Heure de la dernière opération semop()
sem_ctime	time_t	Heure de la dernière modification de sem_perm
sem_nsems	unsigned short	Nombre de sémaphores dans l'ensemble

Lorsqu'un ensemble de sémaphores est créé, les compteurs sont initialement vides. Aucun processus ne peut donc se les attribuer. Il faut donc leur donner une valeur initiale à l'aide de la commande SETALL. En général, on vérifie auparavant si le jeu existe déjà ainsi :

```
if ((sem = semget(cle, nb_sem, 0)) == -1) && (errno == ENOENT)) {
    /* L'ensemble n'existe pas */
    if ((sem = semget(cle, nb_sem, IPC_CREAT | IPC_EXCL | 0600)) == -1) {
        /* Pas assez de mémoire par exemple */
```

```
        perror("semget");
        exit(EXIT_FAILURE);
    }
    /* Maintenant qu'il est créé, on initialise les compteurs */
    for (i = 0; i < nb_sem; i ++)
        table_sem[i]= 1;
    semun.table = table_sem;
    if (semctl(sem, 0, SETALL, semun) < 0)
        perror("semctl");
}
/* L'ensemble peut à présent être utilisé */
```

Un dernier mot avant de présenter un exemple complet d'emploi des sémaphores, pour préciser l'utilisation de l'opération `semop()` avec une valeur `sem_op` nulle. Cela sert, nous l'avons dit, à synchroniser des processus. En fait, on emploie surtout cette opération pour exécuter *N* processus, puis pour attendre qu'ils aient tous atteint un point particulier de leur déroulement avant de les laisser continuer. Pour cette technique de rendez-vous, on utilise un sémaphore qu'on initialise à la valeur *N*. Chaque processus exécute sa première partie, puis arrivé au point de rendez-vous dans son code, il invoque *P()* pour décrémenter la valeur du compteur, suivi d'une opération d'attente avec `sem_op` nulle. Lorsque tous les processus seront arrivés à leurs points de rencontre respectifs, le compteur aura été diminué de *N*, et atteindra donc zéro. Ils pourront alors continuer leur exécution, notamment en appelant *V()* pour restaurer le compteur du sémaphore. En résumé on a :

```
    /* Création d'un sémaphore */
    if ((sem = semget(cle, 1, IPC_CREAT | IPC_EXCL | 0600)) == -1) {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    /* Initialisation */
    table_sem[0]= N;
    semun.table = table_sem;
    if (semctl(sem, 0, SETALL, semun) < 0)
        perror("semctl");
}

    /* Départ des processus */
    ... fork() ...

    /* Point de rendez-vous */
    /*  P()  */
    sembuf.sem_num = 0;
    sembuf.sem_op  =-1;
    sembuf.sem_flg = 0
    semop(sem, & sembuf, 1);
    /*  Attente  */
    sembuf.sem_op  = 0;
    semop(sem, & sembuf, 1);
    /* Rendez-vous Ok */
    /*  V()  */
    sembuf.sem_op  = 1;
    semop(sem, & sembuf, 1);

    /* Suite */
```

Ce procédé fonctionne, car le noyau réveille tous les processus en attente quand le compteur arrive à zéro, avant que les opérations *V()* suivantes ne modifient à nouveau le sémaphore.

Il manque dans ce programme la vérification d'erreur sur `semop()`, surtout en ce qui concerne la destruction du sémaphore, alors que le processus est en attente dessus.

Nous allons à présent examiner un exemple plus complet puisqu'il emploie un sémaphore pour autoriser l'accès à une zone de mémoire partagée. Nous créerons un programme écrivain qui permettra de saisir une chaîne de caractères et de l'écrire dans le segment de mémoire partagée, et un processus lecteur qui affichera l'état de cette mémoire.

Notre processus écrivain maintiendra le sémaphore pendant la saisie, ce qui permettra de le bloquer aussi longtemps que nous le désirerons et de lancer d'autres processus sur un terminal distinct.

exemple_shmwrite.c :

```
    #include <stdio.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/sem.h>
    #include <sys/shm.h>

    #define LG_CHAINE  256

    typedef union semun {
        int                 val;
        struct   semid_ds * buffer;
        unsigned short int * table;
    } semun_t;

    int
main (int argc, char * argv[])
{
    key_t         key;
    int           sem;
    int           shm;
    struct sembuf sembuf;
    semun_t       u_semun;
    char *        chaine = NULL;
    unsigned short table[1];

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier_clé \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((key = ftok(argv[1], 0)) == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    if ((shm = shmget(key, LG_CHAINE, IPC_CREAT | 0600)) == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
```

```
    if ((chaine = shmat(shm, NULL, 0)) == NULL) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    if ((sem = semget(key, 1, 0)) == -1) {
        if ((sem = semget(key, 1, IPC_CREAT | IPC_EXCL | 0600)) == -1) {
            perror("semget");
            exit(EXIT_FAILURE);
        }
        chaine[0] = '\0';
        table[0] = 1;
        u_semun.table = table;
        if (semctl(sem, 0, SETALL, u_semun) < 0)
            perror("semctl");
    }

    sembuf.sem_num = 0;
    sembuf.sem_op  = -1;
    sembuf.sem_flg = SEM_UNDO;
    if (semop(sem, & sembuf, 1) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "> ");
    fgets(chaine, LG_CHAINE, stdin);

    sembuf.sem_op = 1;
    if (semop(sem, & sembuf, 1) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Le programme lecteur attache le segment partagé en lecture seule.

exemple_shmread.c :

```
    #include <stdio.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/sem.h>
    #include <sys/shm.h>

    int
main (int argc, char * argv[])
{
    key_t    key;
    int      sem;
    int      shm;
    struct sembuf sembuf;
    char *   chaine = NULL;
```

```
    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier_clé \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((key = ftok(argv[1], 0)) == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    if (((sem = semget(key, 1, 0)) == -1)
     || ((shm = shmget(key, 0, 0)) == -1) ) {
        perror("semget/shmget");
        exit(EXIT_FAILURE);
    }
    if ((chaine = shmat(shm, NULL, SHM_RDONLY)) == NULL) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    sembuf.sem_num = 0;
    sembuf.sem_op  = -1;
    sembuf.sem_flg = 0;
    if (semop(sem, & sembuf, 1) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "%s\n", chaine);
    sembuf.sem_op = 1;
    if (semop(sem, & sembuf, 1) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Finalement, nous construisons un petit programme de suppression des ressources IPC employées.

exemple_shmctl.c :

```
    #include <stdio.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/sem.h>
    #include <sys/shm.h>

    int
main (int argc, char * argv[])
{
    key_t    key;
    int      sem;
    int      shm;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier_clé \n", argv[0]);
        exit(EXIT_FAILURE);
```

```
    }
    if ((key = ftok(argv[1], 0)) == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    if (((sem = semget(key, 1, 0)) == -1)
     || ((shm = shmget(key, LG_CHAINE, 0)) == -1) ) {
        perror("semget/shmget");
        exit(EXIT_FAILURE);
    }
    shmctl(shm, IPC_RMID, NULL);
    semctl(sem, IPC_RMID, 0);
    return EXIT_SUCCESS;
}
```

Nous utilisons comme fichier-clé l'exécutable `exemple_shmwrite`. Vérifions tout d'abord que les données persistent bien dans un segment partagé après l'arrêt du processus écrivain :

```
$ ./exemple_shmwrite exemple_shmwrite
> Chaîne numéro 1
$ ./exemple_shmread exemple_shmwrite
Chaîne numéro 1
$
```

Nous allons à présent lancer le processus lecteur sur un second terminal, alors que les ressources sont encore tenues par l'écrivain :

```
$ ./exemple_shmwrite exemple_shmwrite
                                        $ ./exemple_shmread exemple_shmwrite
> Chaîne numéro 2
$
                                        Chaîne numéro 2
                                        $
```

Il n'est pas facile de rendre compte ici de l'ordre d'exécution des opérations sur deux terminaux, aussi nous encourageons le lecteur à expérimenter lui-même les divers cas de figure. On peut par exemple tuer avec Contrôle-C le processus écrivain alors qu'il attend une saisie, et vérifier qu'il a bien relâché son sémaphore grâce au mécanisme d'annulation `SEM_UNDO`, en permettant ainsi au lecteur de s'exécuter :

```
$ ./exemple_shmwrite exemple_shmwrite
>  (Contrôle-C)
$ ./exemple_shmread exemple_shmwrite
Chaîne numéro 2
$
```

En fin de compte nous supprimons les ressources alors qu'un processus les emploie :

```
$ ./exemple_shmwrite exemple_shmwrite
                                    $ ./exemple_shmctl exemple_shmwrite
                                    $
> Chaîne numéro 3
semop: Paramètre invalide
$
```

Nous voyons que le jeu de sémaphore a été détruit immédiatement, ce qui déclenche l'erreur sur `semop()`. Par contre, le segment de mémoire partagé n'est supprimé que lorsque le dernier processus a terminé d'y faire référence. Ceci est heureux car nous aurions reçu sinon un signal de faute de segmentation alors que nous utilisions la chaîne de caractères.

Conclusion

Nous l'avons déjà dit, les IPC Système V font souvent office de parents pauvres dans les fonctionnalités standard disponibles sur les machines Unix. Reconnaissons que le principe des files de messages n'est pas très performant, comparé par exemple au système des sockets BSD que nous examinerons dans le chapitre 32. Il ne permet pas de fonctionnement en réseau ni d'utilisation des mécanismes de multiplexage comme nous en verrons dans le prochain chapitre.

Toutefois, pour des transferts rapides de gros volumes de données entre processus distincts, le système des segments de mémoire partagée est parfaitement adapté, à condition de bien synchroniser les accès avec un dispositif comme les sémaphores. L'implémentation des sémaphores Système V est puissante, trop peut-être, ce qui conduit à une complexité de l'interface.

Le travail simultané sur des ensembles complets de sémaphores est une possibilité intéressante pour des programmes expérimentaux, mais pour des applications pratiques, surtout dans un contexte professionnel, il est largement conseillé de limiter au maximum le nombre de sémaphores à manipuler simultanément.

30

Entrées-sorties avancées

Nous allons nous intéresser dans ce chapitre à plusieurs mécanismes différents permettant d'améliorer le fonctionnement des entrées-sorties. Nous allons d'abord observer comment permettre aux appels-système `read()` et `write()` de ne plus être bloquants, même s'ils ne peuvent pas accomplir leurs tâches immédiatement.

Nous étudierons ensuite le multiplexage des entrées-sorties. Cette technique, très utilisée dans les communications entre processus et dans la programmation réseau, permet d'attendre simultanément sur plusieurs canaux l'arrivée de données ou la libération d'un descripteur en écriture.

Finalement, nous examinerons une technique qui peut être très performante dans certaines circonstances, reposant sur des entrées-sorties asynchrones par rapport au déroulement du programme. Bien entendu, un certain nombre de précautions devront être prises avec ce procédé.

Entrées-sorties non bloquantes

Nous avons déjà vu dans le chapitre 28 comment employer l'attribut `O_NONBLOCK` lors de l'ouverture d'un descripteur, afin d'éviter de rester bloqué, même si ce descripteur ne permet pas d'effectuer les opérations demandées. Ceci est très utile avec les tubes de communication et devient même indispensable avec les liaisons série sur certaines machines, lorsque le périphérique concerné ne gère pas le signal CD. Nous en reparlerons plus en détail dans le chapitre 33.

Cependant nous n'avons pas pu obtenir du système qu'il nous laisse effectuer des appels-système `read()` ou `write()` non bloquants sur des descripteurs obtenus autrement qu'avec `open()`, comme un tube ou des entrées-sorties standard. Ceci est malgré tout nécessaire dans certains cas, pour vérifier si des données sont arrivées depuis un descripteur correspondant par exemple au clavier, tout en continuant de mettre à jour régulièrement des informations à l'écran. Notons que cet exemple n'est peut-être pas très judicieux car le clavier, comme nous

l'avons déjà remarqué dans le chapitre 10, ne permet pas dans sa configuration par défaut de capturer des caractères au vol sans appuyer sur la touche Entrée. Nous verrons aussi dans le chapitre 33 comment résoudre ce problème.

Quoi qu'il en soit, nous aimerions disposer de la faculté de lire les données sur un descripteur, sans que cela nous bloque si rien n'est disponible, ou d'écrire dans un tube avec une fonction qui nous signale simplement une erreur si le tube est plein. Ceci est possible en modifiant le comportement du descripteur grâce à l'appel-système **`fcntl()`** que nous avons déjà rencontré dans le chapitre 19. Rappelons que son prototype est défini dans `<fcntl.h>` ainsi :

```
int fcntl (int fd, int commande, ...);
```

Ici la commande employée est celle qui modifie l'attribut du fichier `F_SETFL`, et nous insérons dans le troisième argument l'attribut `O_NONBLOCK`. Dans notre premier exemple, nous allons reprendre le principe d'un programme que nous avons utilisé dans le chapitre 28, en remplissant entièrement un tube. Par contre, nous basculons le descripteur de l'entrée du tube en écriture non bloquante. Ainsi, les appels `write()` après la saturation du buffer (de dimension `PIPE_BUF`, soit 4 096 sur un PC) échoueront sans rester bloqués. Lorsqu'une tentative d'écriture échoue, on endort le processus pendant une seconde pour éviter de consommer inutilement du temps CPU.

exemple_nonblock_1.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    int    tube[2];
    char   c = 'c';
    int    i;

    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    fcntl(tube[1], F_SETFL, fcntl(tube[1], F_GETFL) | O_NONBLOCK);
    i = 0;
    while (1) {
        if (write(tube[1], & c, 1) != 1) {
            perror("write");
            sleep(1);
        } else
            i ++;
        fprintf(stdout, "%d octets écrits \n", i);
    }
    return EXIT_SUCCESS;
}
```

L'exécution illustre bien le comportement attendu. On arrête le programme en interrompant sa boucle avec Contrôle-C.

```
$ ./exemple_nonblock_1
1 octets écrits
2 octets écrits
3 octets écrits
  [...]
4094 octets écrits
4095 octets écrits
4096 octets écrits
write: Ressource temporairement non disponible
4096 octets écrits
write: Ressource temporairement non disponible
4096 octets écrits
write: Ressource temporairement non disponible
4096 octets écrits
write: Ressource temporairement non disponible

  (Contrôle-C)
$
```

Le message « `Ressource temporairement non disponible` » correspond à l'erreur `EAGAIN`, qui indique que l'écriture est momentanément impossible en attendant que le buffer soit vidé par la sortie du tube.

Dans ce programme, nous lisons d'abord la configuration du descripteur avec la commande `F_GETFL` de `fcntl()`, avant de la modifier en ajoutant l'attribut O_NONBLOCK. Cette méthode est préférable, mais on rencontre des applications qui installent directement l'attribut `O_NONBLOCK` sans se soucier de l'état précédent. Le nombre d'attributs étant très restreint (`O_NONBLOCK`, `O_APPEND`, `O_SYNC`), certains programmeurs manipulent directement la configuration globale sans travailler option par option.

Dans notre second exemple, nous allons examiner la lecture non bloquante. Un processus va se scinder en deux, le processus fils assurant une écriture toutes les 700 millisecondes dans un tube. Le père essaye de lire toutes les 100 millisecondes, aussi certaines lectures réussissent-elles quand des données sont disponibles, tandis que d'autres échouent.

exemple_nonblock_2.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    int
main (void)
{
    int   tube[2];
    char  c;

    if (pipe(tube) !=0) {
        perror("pipe");
        exit(EXIT_FAILURE);
```

```
    }
    switch (fork()) {
        case -1 :
            perror("fork");
            exit(EXIT_FAILURE);
        case 0 : /* fils : écriture */
            close(tube [0]);
            while(1) {
                write(tube[1], & c, 1);
                usleep(700000);
            }
        default : /* père : lecture */
            close(tube[1]);
            fcntl(tube[0], F_SETFL,
                   fcntl(tube[0], F_GETFL) | O_NONBLOCK);
            while (1) {
                if (read (tube[0], & c, 1) == 1)
                    printf("Ok \n");
                else
                    printf("Non \n");
                usleep(100000);
            }
    }
    return EXIT_SUCCESS;
}
```

Le programme assure bien une tentative régulière de lecture sans se laisser perturber par ses échecs ni par ses succès.

```
$ ./exemple_nonblock_2
Non
Ok
Non
Non
Non
Non
Non
Ok
Non
Non
Non
Non
Non
Ok
Non
Non
Non
Non
Non
Non
Ok
Non
```

```
   (Contrôle-C)

$
```

Nous n'avons pas affiché de message d'erreur complet, mais la fonction `read()` renvoie aussi le code `EAGAIN` dans `errno` quand une lecture ne donne rien sur un descripteur non bloquant. Le comportement des opérations `read()` et `write()` non bloquantes est le suivant :

Appel	Situation du descripteur	Résultat
`read()`	Aucune donnée disponible	retour : zéro, `errno` = `EAGAIN`
	Moins de données disponibles que le nombre demandé	retour : quantité lue
	Autant ou plus de données disponibles que le nombre demandé	retour : quantité demandée
`write()`	Pas de place	retour : zéro, `errno` = `EAGAIN`
	Pas assez de place pour toute la quantité à écrire	retour : quantité écrite
	Suffisamment de place pour toute l'écriture	retour : quantité demandée

À ceci s'ajoutent bien entendu les valeurs de retour inférieures à zéro (normalement –1) qui correspondent à des erreurs d'entrée-sortie de bas niveau. Dans nos programmes précédents, nous écrivions un seul caractère à la fois, mais en réalité on doit généralement écrire quelque chose comme :

```
    int retour;
    retour = read(fd, buffer, taille);
    if (retour == -1) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    if (retour!= 0)
        traite_donnees_dans_buffer(buffer, taille);
```

Il faut noter que le fait d'ouvrir un descripteur avec l'attribut `O_NONBLOCK` est parfois indispensable, alors que nous voudrions par la suite que les lectures et écritures soient bloquantes. C'est le cas par exemple lors de l'ouverture des deux extrémités d'un tube nommé dans le même processus. Dans cette situation, on utilise alors un arrangement comme celui-ci :

```
    fd = open(nom_du_fichier, O_RDONLY | O_NONBLOCK);
    if (fd >= 0) {
        fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) & (~O_NONBLOCK));
        ...
    }
```

Avec les lectures et écritures non bloquantes, nous pouvons déjà écrire des applications avec un comportement assez dynamique, dont le déroulement continue imperturbablement, que des données arrivent ou non. Voici par exemple un squelette de jeu dans lequel on lit le clavier de manière non bloquante :

```
    int
main (void)
{
    char touche; /* Pan ! */
```

```
    fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
    while (1) {
        if (calcul_des_nouveaux_evenements() == FIN_PARTIE)
            break;
        if (read(STDIN_FILENO, & touche, 1) == 1)
            calcul_deplacement_joueur(touche);
        affichage_nouvelle_situation();
        usleep(UN_25EME_DE_SECONDE);
    }
    affichage_du_score();
    return EXIT_SUCCESS;
}
```

De même, les écritures non bloquantes peuvent être très utiles lorsque l'émission de données vers un autre processus est une fonctionnalité annexe ne devant en aucun cas freiner le cours de l'application principale. Ceci concerne par exemple les sorties de journalisation d'un système temps-réel – sauf bien entendu lorsque cette journalisation est considérée comme un mécanisme de boîte noire dont l'importance est cruciale pour le suivi de l'application.

Dans le cas d'un processus serveur ayant des connexions par tube – ou socket réseau – avec de multiples clients, on pourrait être tenté d'écrire une boucle principale comme :

```
    while (1) {
        for (i = 0; i < nombre_de_clients; i ++)
            if (read(tube_depuis_client[i], & requete, sizeof(requete_t))
                     == sizeof(requete_t))
                repondre_a_la_requete(i, & requete);
    }
```

Toutefois ce code serait très mauvais, car il accomplirait la plupart du temps des boucles vides, consommant inutilement et exagérément du temps processeur. La seule situation où on pourrait tolérer ce genre de comportement serait dans des portions *courtes* de logiciel temps-réel, où on attend simultanément des messages sur plusieurs canaux de communication, et quand la réponse doit être fournie dans un délai ne tolérant pas le risque que le processus soit endormi temporairement. Hormis ce cas très particulier, on se tournera plutôt vers le mécanisme d'attente passive permettant un multiplexage des entrées.

Attente d'événements – Multiplexage d'entrées

On a souvent besoin, principalement dans les applications de serveur réseau, de surveiller l'arrivée de données en provenance de multiples sources. Mais étant donné que toutes les entrées-sorties ont lieu sous le contrôle du noyau, il est généralement inutile d'effectuer des boucles d'attente active comme nous l'avons vu à la section précédente. L'appel-système `select()` – apparu en 1982 dans BSD 4.2 – et l'appel `poll()` – intégré à Système V R 3 en 1986 – permettent de dire en substance au noyau : « Voici la liste des descripteurs qui m'intéressent, préviens-moi s'il se passe quelque chose, en attendant je fais un petit somme. » L'application relâche entièrement le processeur, au bénéfice des autres programmes. Quand des données arrivent, quel que soit le type de descripteur, elles passent par le noyau, qui se souvient alors qu'un processus est en attente et peut le réveiller en lui indiquant que des informations sont prêtes à être lues.

Ces appels-système sont essentiels dans de nombreuses situations. Ils sont décrits SUSv3. L'appel **select()** est plus employé que poll() même si sa syntaxe est plus compliquée, aussi le décrirons-nous en premier. Il est déclaré dans <sys/types.h> ainsi :

```
int select(int nb_descripteurs,
           fd_set * ensemble_a_lire,
           fd_set * ensemble_a_ecrire,
           fd_set * ensemble_exceptions,
           struct timeval * delai_maxi);
```

Il prend en arguments trois pointeurs sur des ensembles de descripteurs, un pointeur NULL correspondant à un ensemble ignoré :

- Le premier ensemble est surveillé par le noyau en attente de données à lire. Dès que des informations sont disponibles, le processus est réveillé. Nous étudierons d'abord ce principe.
- Les descripteurs du second ensemble correspondent à des sorties du processus. On désire ici qu'un de ces descripteurs accepte de recevoir des données. On attend par exemple qu'un buffer se vide en libérant de la place ou qu'un processus lecteur ait ouvert un tube nommé.
- Le troisième ensemble est rarement utilisé car il contient des descripteurs sur lesquels on attend l'arrivée de conditions exceptionnelles. Ceci correspond généralement à l'arrivée de données urgentes hors bande sur une socket réseau TCP.

L'appel-système select() permet aussi de configurer un délai d'attente maximal. Lorsque celui-ci est écoulé, le noyau termine l'appel avec un code de retour nul. Cette fonctionnalité a longtemps été utilisée pour endormir un processus, de manière portable, avec une meilleure précision que la seconde. En effet, l'appel select() est plus répandu que nanosleep() et offre, grâce à la structure timeval, la possibilité de configurer un sommeil avec une résolution de l'ordre de la microseconde. On emploie un code du genre :

```
struct timeval attente;
attente.tv_sec  = delai_en_microsecondes / 1000000;
attente.tv_usec = delai_en_microsecondes % 1000000;
select(0, NULL, NULL, NULL, & attente);
```

Lorsque le pointeur sur la structure timeval est NULL, l'appel-système reste bloqué indéfiniment en attente d'une condition favorable sur un descripteur. Si, au contraire, la valeur du délai vaut 0, l'appel revient immédiatement sans bloquer.

Le premier argument de select() est finalement le plus compliqué. Il s'agit du numéro du plus grand descripteur de fichier contenu dans les ensembles surveillés, augmenté de 1. Ceci sert au noyau pour dimensionner un masque de bits lui indiquant en interne quels descripteurs surveiller. Pour positionner cette valeur, on peut employer par exemple :

```
plus_grand_descripteur = -1;
for (i = 0; i < nombre_de_descripteurs; i ++)
    if (descripteur[i] > plus_grand_descripteur)
        plus_grand_descripteur = descripteur[i];
select(plus_grand_descripteur + 1, ...);
```

Sachant que le noyau fournit les descripteurs de fichiers dans l'ordre, en commençant par 0, 1 et 2, qui sont attribués aux flux standard d'entrée, de sortie et d'erreur, on peut aussi avoir

recours à quelques astuces, dont la plus courante est d'employer la constante symbolique **FD_SETSIZE**, qui correspond à la taille maximale d'un ensemble de descripteurs.

Ces ensembles sont du type opaque `fd_set`. On les manipule par le biais des macros suivantes :

```
FD_ZERO  (fd_set * ensemble);
FD_SET   (int fd, fd_set * ensemble);
FD_CLR   (int fd, fd_set * ensemble);
FD_ISSET (int fd, fd_set * ensemble);
```

La macro **FD_ZERO()** permet d'initialiser un ensemble vide. Il faut toujours l'employer au début de l'utilisation d'un ensemble. La macro **FD_SET()** ajoute un descripteur dans un ensemble, tandis que **FD_CLR()** en supprime un. Enfin, **FD_ISSET()** permet de vérifier si un descripteur est présent ou non dans un ensemble.

En effet, au retour de `select()`, l'appel-système renvoie le nombre de descripteurs se trouvant dans les conditions attendues et modifie les ensembles passés en arguments, pour n'y laisser que les descripteurs concernés. Si l'appel-système `select()` est interrompu par un signal, il renvoie –1 et configure `EINTR` dans `errno`.

Naturellement, on utilise `select()` uniquement sur des descripteurs correspondant à des tubes, des FIFO, des fichiers spéciaux de périphériques ou des sockets réseau. Si toutefois on transmet un descripteur de fichier régulier, `select()` considère que des données sont disponibles tant qu'on n'est pas arrivé à la fin du fichier.

En oubliant pour le moment les descripteurs en attente d'écriture et de conditions exceptionnelles, nous pouvons considérer le multiplexage de plusieurs lectures :

```
    int
attente_reception (int descripteurs[],
                   int nb_descripteurs,
                   int delai_maxi)
{
    struct timeval attente;
    fd_set         ensemble;
    int            plus_grand = -1;
    int            i;
    int            retour;

    attente.tv_sec  = delai_maxi;
    attente.tv_usec = 0;
    /* initialisation de l'ensemble */
    FD_ZERO(& ensemble);
    for (i=0; i < nb_descripteurs; i ++) {
        if (descripteur[i] > FD_SETSIZE) {
            fprintf(stderr, "Descripteur trop grand \n") ;
            return -1;
        }
        FD_SET(descripteur[i], & ensemble);
        if (descripteur[i] > plus_grand)
            plus_grand = descripteur[i];
    }
    /* attente */
    do {
```

```
        retour = select(plus_grand + 1,
                        & ensemble, NULL, NULL,
                        & attente);
    } while ((retour == -1) && (errno == EINTR));
    if (retour < 0) {
        perror("select");
        return -1;
    }
    if (retour == 0) {
        fprintf(stderr, "délai dépassé \n");
        return -1;
    }
    /* examen des descripteurs prêts */
    for (i = 0; i < nb_descripteurs ; i ++)
        if (FD_ISSET(descripteur[i], & ensemble))
            lecture_descripteur(descripteur[i]);
    return 0;
}
```

Nous contrôlons que les descripteurs ont bien une valeur inférieure à `FD_SETSIZE`. C'est une attitude vraiment paranoïaque, ayant rarement cours dans les applications courantes, le risque de dépasser cette valeur étant infime (sauf si on ouvre à répétition un descripteur en oubliant de le refermer).

Dans le programme suivant, nous créons 10 fils et 10 tubes de communication avec leur processus père. Ce dernier va surveiller les arrivées avec `select()`. Les 10 fils enverront régulièrement un caractère à leur père, chacun avec une fréquence différente variant entre une fois par seconde et une fois toutes les dix secondes. Il manque de nombreuses vérifications d'erreur, qui auraient alourdi inutilement le listing.

exemple_select.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <sys/types.h>

    #define    NB_FILS    10

    int
main (void)
{
    int      tube[NB_FILS][2];
    fd_set   ensemble;
    int      i, fils;
    char     c = 'c';

    for (i = 0; i < NB_FILS; i ++)
        if (pipe(tube[i]) < 0) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
    for (fils = 0; fils < NB_FILS; fils ++)
```

```
        if (fork() == 0)
            break;
    for (i = 0; i < NB_FILS; i ++)
        if (fils == NB_FILS) {
            /* On est dans le père */
            close(tube[i][1]);
        } else {
            close(tube[i][0]);
            if (i != fils)
                close(tube[i][1]);
        }
    if (fils == NB_FILS) {
        while (1) {
            FD_ZERO(& ensemble);
            for (i = 0; i < NB_FILS; i ++)
                FD_SET(tube[i][0], & ensemble);
            if (select(FD_SETSIZE, & ensemble, NULL, NULL, NULL) < 0){
                perror("select");
                break;
            }
            for (i = 0; i < NB_FILS; i ++)
                if (FD_ISSET(tube[i][0], & ensemble)) {
                    fprintf(stdout, "%d ", i);
                    fflush(stdout);
                    read(tube[i][0], & c, 1);
                }
        }
    } else { /* On est dans un fils */
        while (1) {
            usleep((fils + 1) * 1000000);
            write(tube[fils][1], & c, 1);
        }
    }
    return 0;
}
```

Lors de son exécution, ce programme adopte bien le comportement dynamique qu'on attend, tout en évitant de faire des boucles actives consommatrices inutiles de ressources processeur.

```
$ ./exemple_select
0 1 0 2 0 3 1 0 4 0 5 2 1 0 6 0 7 3 1 0 8 2 0 9 4 1 0
  (Contrôle-C)
$
```

Lorsqu'un appel-système `select()` se termine avant que le délai maximal soit écoulé, soit parce qu'un descripteur est prêt, soit parce qu'un signal l'a arrêté, le noyau Linux modifie le contenu de la structure `timeval` passée en dernier argument afin qu'elle contienne la durée restante non écoulée. Ce comportement est bien commode dans certains cas, notamment lorsqu'on implémente un bouclage pour ignorer les interruptions dues aux signaux :

```
    do {
        retour = select (FD_SETSIZE, & ensemble, NULL, NULL, & attente);
    } while ((retour == -1) && (errno == EINTR));
```

Toutefois, il faut savoir que ce comportement n'est pas portable. La plupart des autres Unix de la famille Système V ne modifient pas ce délai. Le même problème peut se poser pour porter sous Linux une application qui considère que cette variable n'est pas modifiée et la réutilise directement. Pour pallier ce problème, le noyau Linux offre la possibilité de modifier la *personnalité* du processus grâce à l'appel-système **`personality()`**. Ce dernier permet de demander au noyau d'adopter, avec ce processus, une attitude différente dans certains appels-système, ainsi que de numéroter autrement les signaux. Nous ne détaillerons pas cette fonction car elle est très spécifique et peu recommandée. Il vaut mieux corriger l'application défectueuse que de demander au noyau d'émuler les bogues des autres systèmes.

Pour revenir au problème de la modification du délai, l'attitude la plus prudente consiste à ne pas faire de supposition concernant l'état de la variable et à la remplir à nouveau avant chaque appel, en calibrant la nouvelle attente avec `gettimeofday()`.

Une variante de `select()`, nommée **`pselect()`** est disponible, mais moins utilisée :

```
int select(int nb_descripteurs,
           fd_set * ensemble_a_lire,
           fd_set * ensemble_a_ecrire,
           fd_set * ensemble_exceptions,
           struct timeval * delai_maxi,
           const sigset_t * signaux);
```

Elle prend un argument supplémentaire par rapport à `select()` : c'est le masque des signaux qui sera utilisé durant l'attente. Ceci permet d'attendre de manière fiable et simultanée l'occurrence de signaux (que l'on bloque en dehors de l'appel système pour ne pas risquer d'en perdre) et de situations sur des descripteurs.

Il existe un autre appel de multiplexage, **`poll()`**, issu de l'univers Système V, déclaré dans `<poll.h>` :

```
int poll (struct pollfd * pollfd,
          unsigned int nb_structures,
          int delai_maxi);
```

Les descripteurs à surveiller sont indiqués dans une table de structures `pollfd`, contenant les membres suivants :

Nom	Type	Signification
`fd`	`Int`	Descripteur de fichier à surveiller.
`events`	`short int`	Liste des événements qui nous intéressent concernant ce descripteur.
`revents`	`short int`	Ensemble des événements survenus, au retour de l'appel-système.

Le noyau examine donc les événements attendus sur chaque descripteur et modifie le membre `revents` avant de revenir de l'appel-système. Les événements qui peuvent survenir sont les suivants :

Nom	Signification
`POLLIN`	Données disponibles pour la lecture
`POLLOUT`	Descripteur prêt à recevoir des données

Nom	Signification
POLLPRI	Données urgentes disponibles (informations hors bande sur connexion TCP)
POLLERR	Erreur survenue sur le descripteur (uniquement en réponse dans revents)
POLLHUP	Déconnexion d'un correspondant (uniquement en réponse dans revents)
POLLNVAL	Descripteur invalide (uniquement en réponse dans revents)

Comme select(), poll() renvoie normalement le nombre de structures pour lesquelles il s'est passé quelque chose (éventuellement une erreur) et une valeur nulle si le délai est dépassé. S'il est interrompu par un signal, poll() renvoie –1 et positionne errno avec EINTR.

L'avantage de poll() par rapport à select() c'est qu'il n'y a pas de limite absolue au nombre de descripteurs surveillés simultanément. Par contre, cette fonction est nettement moins répandue que select() dans les Unix de la famille BSD. Les programmeurs préfèrent donc généralement employer select().

Distribution de données – Multiplexage de sorties

Le multiplexage de données en sortie est plus rare, car un processus qui veut envoyer des données à un correspondant préfère souvent rester bloqué quelque temps mais être sûr que ses informations sont émises. Il est toutefois possible d'avoir à écrire un volume important de données sur plusieurs descripteurs simultanément. On peut alors implémenter un système de mémoire tampon, avec lequel on écrit avec des write() non bloquants sur les descripteurs qui sont prêts.

On peut aussi cumuler un multiplexage d'entrées et de sorties dans le même appel select(). Supposons qu'on reçoive de manière continue des données provenant d'un tube source et qu'on doive les distribuer autant que possible vers des tubes cibles. On pourrait utiliser un buffer de sortie pour chaque cible. Une implémentation pourrait être :

```
    int
distribution (int source, int cibles [], int nb_cibles)
{
    fd_set   ensemble_lecture;
    fd_set   ensemble_ecriture;
    int      i;
    char **  buffer_cible = NULL;
    int *    contenu_buffer = NULL;
    char     buffer_source[LG_BUFFER];
    int      contenu_source;
    int      nb_ecrits;

    /* Allouer un tableau de buffers (1 pour chaque source) */
    /* avec un indicateur du contenu pour chaque buffer     */
    if ((contenu_buffer = calloc(nb_cibles, sizeof(int))) == NULL)
        return -1;
    if ((buffer_cible = calloc(nb_cibles, sizeof(char *))) == NULL) {
        free(contenu_buffer);
        return -1;
    }
```

```
    /* Allouer les buffers proprement dits                  */
    for (i = 0; i < nb_cibles; i ++) {
        if ((buffer_cible[i] = malloc(LG_BUFFER)) == NULL) {
            while (--i >= 0)
                free(buffer_cible[i]);
            free(buffer_cible);
            free(contenu_buffer);
            return -1;
        }
        contenu_buffer[i] = 0;
    }
    while (1) {
        FD_ZERO(& ensemble_lecture);
        FD_SET(source, & ensemble_lecture);
        FD_ZERO(& ensemble_ecriture);
        for (i = 0; i < nb_cibles; i ++)
            if (contenu_buffer[i] > 0)
                /* Il reste des données à écrire sur cette cible */
                FD_SET(cibles[i], & ensemble_ecriture);
        /* Attendre l'arrivée de données où la libération     */
        /* d'une cible vers laquelle il reste des données à écrire */
        while (select(FD_SETSIZE,
                      & ensemble_lecture, & ensemble_ecriture, NULL,
                      NULL) < 0)
            if (errno != EINTR)
                perror("select");

        if (FD_ISSET(source, & ensemble_lecture)) {
            /* Il y a des données à lire */
            contenu_source = read(source, buffer_source, LG_BUFFER);
            if (contenu_source > 0)
                /* On les transmet dans les buffers des cibles */
                for (i = 0; i < nb_cibles; i ++)
                    if (contenu_source + contenu_buffer [i] < LG_BUFFER) {
                        /* Il y a assez de place sur cette cible */
                        memcpy (& (buffer_cible[i][contenu_buffer[i]]),
                                buffer_source,
                                contenu_source);
                        contenu_buffer[i] += contenu_source;
                    } else {
                        /* la cible ne lit pas assez vite, on copie ce */
                        /* qu'on peut.                              */
                        memcpy (&(buffer_cible[i][contenu_buffer[i]]),
                                buffer_source,
                                LG_BUFFER - contenu_buffer[i]);
                        contenu_buffer[i] = LG_BUFFER;
                    }
        }
        for (i = 0; i < nb_cibles; i ++) {
            if (FD_ISSET(buffer_cible[i], & ensemble_ecriture)) {
                /* Il y a de la place libérée sur la cible */
```

```
                nb_ecrits = write(cibles[i],
                                  buffer_cible[i],
                                  contenu_buffer[i]);
                if (nb_ecrits > 0) {
                    memmove(buffer_cible[i],
                            & (buffer_cible[i][nb_ecrits]),
                            contenu_buffer[i] - nb_ecrits);
                    contenu_buffer[i] -= nb_ecrits;
                }
            }
        }
    }
}
```

On imagine bien la puissance de l'appel-système `select()`, puisqu'il permet de centraliser totalement la surveillance de multiples canaux de communication, l'application étant prête à réagir immédiatement lorsque des données arrivent ou dès qu'un descripteur de sortie est libre.

Ce mécanisme est particulièrement précieux dès qu'une application doit se comporter en serveur vis-à-vis de multiples clients, mais il est parfois insuffisant. Une application fonctionnant sous X-Window par exemple doit invoquer lors de son initialisation une routine de boucle principale fournie par les bibliothèques graphiques, généralement `XtAppMainLoop()`, qui ne lui rend pas la main mais invoquera les fonctions *callback* associées aux composants graphiques lors de leurs activations. Un tel programme ne peut donc pas rester bloqué sur `select()`.

Une solution possible est d'employer un `fork()` avant l'initialisation graphique, pour laisser le processus fils s'occuper de tout le dialogue avec les clients alors que le processus père ne gère que l'interface graphique. La communication entre les processus est réduite au minimum, et s'établit au moyen de deux tubes et de signaux.

Nous pouvons également utiliser le mécanisme des entrées-sorties asynchrones offert par l'appel-système `fcntl()`.

Entrées-sorties asynchrones avec fcntl()

L'appel-système `fcntl()`, que nous avons déjà rencontré dans le chapitre 19, permet d'implémenter un mécanisme d'entrées-sorties asynchrones assez intéressant car il offre un multiplexage des entrées ou des sorties comme avec `select()`, tout en laissant le processus libre d'exécuter du code au lieu de le bloquer en attente.

Ce système est plutôt réservé aux tubes de communication ou aux sockets. Le principe consiste à associer au descripteur de fichier le PID du processus, et de laisser le noyau nous prévenir lorsque des données seront disponibles en lecture ou lorsqu'il y aura à nouveau de la place pour écrire dans le descripteur. Dès que l'une de ces conditions se présente, le noyau nous envoie le signal `SIGIO`.

Nous avons donc une sorte d'équivalent de `select()`, puisque nous serons prévenus lorsqu'une condition d'entrée-sortie sera réalisée, tout en conservant la liberté d'exécuter le reste du programme.

Les commandes employées avec `fcntl()` sont les suivantes :

- **`F_SETOWN`** : on indique le PID du processus devant recevoir le signal. Un PID négatif correspond à un groupe de processus.

  Normalement, on utilise `fcntl (fd, F_SETOWN, getpid())`.
- **`F_GETOWN`** : l'appel `fcntl(fd, F_GETOWN)` renvoie le PID du processus ou du groupe récepteur.
- **`F_SETSIG`** : ceci permet de configurer le numéro du signal à employer pour prévenir des modifications de conditions. Par défaut ou en cas de valeur nulle, le signal `SIGIO` est envoyé. Toutefois, nous préférerons utiliser un signal temps-réel, car des informations supplémentaires seront disponibles. Nous pouvons employer un signal différent pour chaque descripteur, quoique cela ne soit pas indispensable avec les signaux temps-réel.

  On appellera donc `fcntl (fd, F_SETSIG, SIGRTMIN + 5)`.
- **`F_GETSIG`** : avec `fcntl(fd, F_GETSIG)`, on peut connaître le numéro de signal employé.

Le gestionnaire de signaux reçoit, s'il s'agit d'un signal temps-réel, une structure `siginfo` dont le champ `si_code` contient la valeur `SI_SIGIO`, et le membre `si_fd` le descripteur du fichier concerné. Si on utilise le signal `SIGIO` d'origine, le gestionnaire ne dispose pas de cette structure `siginfo`, il devra donc employer `select()` avec un délai nul pour déterminer quel descripteur est devenu disponible.

Pour activer le comportement asynchrone, il faut faire appel à la commande `F_SETFL` de `fcntl()` et activer l'attribut `O_ASYNC` ainsi :

```
fcntl (fd, F_SETFL, fcntl (fd, F_GETFL) | O_ASYNC);
```

Notons qu'à la différence de `select()` le noyau nous prévient ici lorsque les conditions de lecture ou d'écriture sont modifiées. Si des données sont disponibles en lecture avant l'activation de `O_ASYNC` ou s'il y a déjà de la place pour l'écriture, nous n'en serons pas prévenus. Une solution consiste à s'envoyer systématiquement un signal juste après le basculement en mode asynchrone et à vérifier au sein du gestionnaire l'état du descripteur, à l'aide d'un appel `select()` avec un délai nul.

Ces possibilités sont intéressantes, malheureusement de nombreuses versions du noyau Linux (y compris dans les branches 2.4 et 2.6) n'implémentent pas correctement ces fonctionnalités, et il est difficile d'obtenir un programme qui s'exécute bien sur les différents noyaux disponibles. Toutefois nous pourrons résoudre ce problème en nous tournant vers un autre mécanisme – totalement portable celui-ci : l'asynchronisme introduit par la norme Posix.1b. et défini de nos jours par SUSv3.

Entrées-sorties asynchrones Posix.1b

Jusqu'à présent nous avons réussi à optimiser la communication sur plusieurs canaux simultanément grâce au multiplexage de `select()`, et à laisser le programme se dérouler normalement en réitérant de temps à autre les tentatives d'entrée-sortie.

Toutefois ces mécanismes ne sont pas suffisants dans le cas où on désire vraiment recevoir ou émettre des données de manière suffisamment fiable, avec un mode opératoire totalement asynchrone par rapport au reste du programme. En effet, lorsque `select()` – ou un signal programmé par `fcntl()` - nous indique que des données sont disponibles en lecture, tout ce

que nous savons c'est que le descripteur est prêt à nous délivrer un octet. Si nous désirons en lire plusieurs, l'appel `read()` peut être bloquant. Si on bascule en lecture non bloquante, il faut de surcroît gérer un buffer interne pour recevoir assez d'informations avant de les traiter.

Le fonctionnement des écritures n'est pas plus sûr : `select()` nous précise qu'on peut écrire au moins un octet. Maintenant, si on veut transmettre une trame complète de données, il faut s'attendre à ce que `write()`bloque indéfiniment. L'écriture non bloquante nécessiterait d'établir un tampon de sortie pour s'assurer que l'ensemble de données sera écrit.

Heureusement, on peut employer des procédures d'entrées-sorties totalement asynchrones nous épargnant la gestion d'un buffer. On programme une opération de lecture ou d'écriture, le noyau la démarre, et lorsqu'elle est terminée le processus est averti par exemple par l'arrivée d'un signal. Durant le temps de l'opération d'entrée-sortie, le programme est libre de faire ce que bon lui semble, utiliser le processeur, faire des appels-système, dormir…

Une expérience instructive consiste à lancer la commande

```
$ find / -name introuvable
```

tout en surveillant l'activité du processeur, par l'intermédiaire de `top` ou de `xload` par exemple. Cette commande va parcourir toute l'arborescence du système de fichiers à la recherche des nœuds ayant le nom `introuvable`. On observe que le disque est fortement mis à contribution, une activité incessante et prolongée s'y déroulant. Par contre, on remarque que le processeur lui-même reste dans un état calme, sa charge étant très faible. Les procédures d'entrée-sortie utilisent donc très peu de ressources de calcul de la machine.

Dans une application temps-réel, il peut donc être très intéressant de déléguer une part du travail d'enregistrement par exemple, en le laissant s'exécuter automatiquement tandis que l'application peut continuer à répondre aux événements survenant entre-temps.

Les mécanismes d'entrées-sorties asynchrones sont disponibles si la constante **`_POSIX_ASYNCHRONOUS_IO`** est définie dans `<unistd.h>`, ce qui est le cas depuis Linux 2.2.

Jusqu'au noyau Linux 2.4 inclus, ces mécanismes sont en fait des fonctions de la bibliothèque C qui sont implémentées au moyen des threads Posix. L'édition des liens devait se faire avec la bibliothèque `librt.so` (*real time*) et la bibliothèque `libpthread.so` au moyen des options `–lrt –lpthread` en ligne de commande.

Depuis le noyau 2.6, il s'agit véritablement de mécanismes asynchrones implémentés par le noyau, seule l'option `–lrt` reste nécessaire.

Le principe des entrées-sorties asynchrones conformes à la norme SUSv3 n'est guère plus compliqué que celui des entrées-sorties classiques : on prépare un bloc constitué par une structure **`aiocb`**, contenant en substance le buffer, le descripteur de fichier et le type d'opération désirée. Ce bloc est transmis au noyau qui programme l'entrée-sortie, puis prévient le processus par un signal lorsque l'opération est terminée. La structure `aiocb` comprend les membres suivants :

Nom	Type	Signification
`aio_fildes`	`int`	Descripteur du fichier concerné par l'opération d'entrée-sortie.
`aio_offset`	`off_t`	Emplacement au sein du fichier où commence l'opération.
`aio_buf`	`void *`	Buffer pour les données à écrire ou à lire.

Nom	Type	Signification
aio_nbytes	size_t	Nombre d'octets à transférer.
aio_reqprio	int	Priorité de l'opération.
aio_sigevent	struct sigevent	Description du mécanisme de signalisation une fois le transfert terminé.
aio_lio_opcode	int	Code opératoire décrivant le transfert (uniquement dans certains cas).

Pour programmer une lecture ou une écriture, on emploie les fonctions **aio_read()** ou **aio_write()**, déclarées ainsi dans <aio.h> :

```
int aio_read  (struct aiocb * aiocb);
int aio_write (struct aiocb * aiocb);
```

La structure aiocb contient donc le numéro du descripteur de fichier, l'adresse du buffer pour les données et la taille désirée, mais également le décalage où l'opération doit avoir lieu dans le fichier. Ce décalage est mesuré en octets, comme avec lseek(), depuis le début du fichier. En effet, la position courante dans le fichier n'est jamais significative avec les entrées-sorties asynchrones. L'emplacement de lecture ou d'écriture doit toujours être indiqué. Nous reviendrons sur ce point ultérieurement.

Le membre aio_reqprio dispose d'une valeur numérique indiquant la valeur qui doit être soustraite de la priorité du processus pour exécuter l'opération d'entrée-sortie. Ceci n'a d'intérêt que si on déclenche de nombreuses opérations simultanées. Plus cette valeur est élevée, moins l'opération sera prioritaire par rapport à ses consœurs. La priorité de l'opération n'est que rarement utilisée, aussi la remplit-on généralement avec une valeur nulle. Nous en verrons toutefois un exemple d'utilisation plus tard. Avec les appels aio_read() et aio_write(), on n'emploie pas non plus le membre aio_lio_opcode puisque le système sait toujours quelle opération doit avoir lieu.

Pour indiquer que le transfert asynchrone est terminé – avec succès ou non –, le système peut nous envoyer un signal ou démarrer un thread sur une fonction spéciale. Pour configurer ce comportement, on utilise la structure **sigevent** du champ aio_sigevent, définie dans <signal.h> ainsi :

Nom	Type	Signification
sigev_notify	int	Type de notification désirée pour indiquer la fin d'une opération asynchrone
sigev_signo	int	Numéro du signal à employer pour la notification
sigev_value	sigval_t	Valeur à transmettre au gestionnaire de signaux ou au thread
sigev_notify_function	void (* f) (sigval_t)	Fonction à déclencher dans un nouveau thread
sigev_notify_attributes	pthread_attr_t	Attribut du nouveau thread

Attention

Les membres sigev_notify_function et sigev_notify_attributes sont en réalité des macros qui donnent accès aux champs d'une union assez complexe. On évitera donc de nommer ainsi des variables.

Le type **sigval_t** est un autre nom de union sigval, que nous avons rencontrée dans le chapitre 8, et qui peut prendre les formes suivantes :

Nom	Type
sival_int	int
sival_ptr	void *

Le membre sigev_notify contient l'une des constantes symboliques suivantes :

- **SIGEV_NONE** : aucune notification n'est demandée. Le processus pourra toutefois s'assurer de la fin d'une opération en employant des routines que nous décrirons plus bas.
- **SIGEV_SIGNAL** : le système enverra au processus le signal mentionné dans le champ sigev_signo pour indiquer que l'opération est terminée. S'il s'agit d'un signal temps-réel Posix.1b, le gestionnaire recevra dans son argument siginfo_t des informations supplémentaires, dont la valeur du membre sigev_value. Le champ si_code de la structure siginfo_t est rempli avec le code SI_ASYNCIO, comme nous l'avons déjà évoqué dans le chapitre 8.
- **SIGEV_THREAD** : la bibliothèque C démarrera un nouveau thread, qui exécutera la fonction sur laquelle le champ sigev_notify_function représente un pointeur. Cette routine recevra en argument le contenu du membre sigev_value. Le thread créé reçoit les attributs décrits par le champ sigev_notify_attributes. Il s'agit des attributs au sens des Pthreads, comme nous les avons vus dans le chapitre 12 (détachable, joignable, etc.)

Attention

Avec Linux 2.6 la notification par un thread ne fonctionne pas au moment de la rédaction de ces lignes (noyau 2.6.9). Le problème est semble-t-il lié aux interactions avec la GlibC (2.3.2) et devrait être corrigé à l'avenir.

Avec SIGEV_THREAD comme avec SIGEV_SIGNAL, on remplit généralement le membre sival_ptr du champ sigev_value avec un pointeur sur la structure aiocb elle-même, afin que le nouveau thread ou le gestionnaire aient accès à l'opération réalisée. Naturellement, on ne peut pas réemployer la même structure avant que l'opération soit terminée.

Pour savoir si une opération est terminée ou non, on utilise la fonction **aio_error()** :

```
int aio_error (const struct aiocb * aiocb);
```

Cette routine renvoie l'erreur **EINPROGRESS** si l'opération décrite par la structure aiocb n'est pas terminée. Sinon elle transmet éventuellement un indicateur d'erreur. Une fois qu'une opération est finie, et uniquement à ce moment-là, on peut appeler la fonction **aio_return()** pour avoir le compte rendu de l'entrée-sortie :

```
ssize_t aio_return (const struct aiocb * aiocb);
```

Cette fonction renvoie tout simplement la valeur de retour des appels-système read() ou write() sous-jacents. Cette fonction ne doit être appelée qu'une seule fois car Posix.1b autorise une implémentation où elle servirait à libérer des données internes.

Même dans le gestionnaire de signaux servant à la notification, il faut donc employer toujours la séquence :

```
    if (aio_error(aiocb) == EINPROGRESS)
        return;
    if ((retour = aio_return(aiocb)) != aiocb -> aio_nbytes)
        /* Traitement d'erreur */
    else
        /* Réussite */
```

On peut très bien éviter la notification et vérifier plus tard explicitement si l'opération s'est bien terminée. Dans le programme suivant nous allons utiliser trois lectures asynchrones, employant les trois possibilités de notification. La lecture demandée est la même à chaque fois, on réclame les 256 premiers octets du fichier dont le nom est passé en argument.

exemple_aio_read.c :

```
    #include <aio.h>
    #include <errno.h>
    #include <stdio.h>
    #include <signal.h>
    #include <unistd.h>
    #include <sys/stat.h>

    #define SIGNAL_IO    (SIGRTMIN + 3)

    void
gestionnaire (int signum, siginfo_t * info, void * vide)
{
    struct aiocb * cb;
    ssize_t        nb_octets;
    if (info->si_code == SI_ASYNCIO) {
        cb = info->si_value.sival_ptr;
        if (aio_error(cb) == EINPROGRESS)
            return;
        nb_octets = aio_return(cb);
        fprintf(stdout, "Lecture 1 : %d octets lus \n", nb_octets);
    }
}

    void
thread (sigval_t valeur)
{
    struct aiocb * cb;
    ssize_t        nb_octets;
    cb = valeur.sival_ptr;
    if (aio_error(cb) == EINPROGRESS)
        return;
    nb_octets = aio_return(cb);
    fprintf(stdout, "Lecture 2 : %d octets lus \n", nb_octets);
}

    int
main (int argc, char * argv[])
{
```

```
int              fd;
struct aiocb     cb[3];
char             buffer[256][3];
struct sigaction action;
int              nb_octets;

if (argc != 2) {
    fprintf(stderr, "Syntaxe : %s fichier \n", argv [0]);
    exit(EXIT_FAILURE);
}
if ((fd = open(argv[1], O_RDONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
action.sa_sigaction = gestionnaire;
action.sa_flags     = SA_SIGINFO;
sigemptyset(& action.sa_mask);
if (sigaction(SIGNAL_IO, & action, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
/* Lecture 0 :  Pas de notification */
cb[0].aio_fildes   = fd;
cb[0].aio_offset   = 0;
cb[0].aio_buf      = buffer [0];
cb[0].aio_nbytes   = 256;
cb[0].aio_reqprio  = 0;
cb[0].aio_sigevent.sigev_notify = SIGEV_NONE;

/* Lecture 1 : Notification par signal */
cb[1].aio_fildes   = fd;
cb[1].aio_offset   = 0;
cb[1].aio_buf      = buffer [1];
cb[1].aio_nbytes   = 256;
cb[1].aio_reqprio  = 0;
cb[1].aio_sigevent.sigev_notify = SIGEV_SIGNAL;
cb[1].aio_sigevent.sigev_signo  = SIGNAL_IO;
cb[1].aio_sigevent.sigev_value.sival_ptr = & cb[1];

/* Lecture 2 : Notification par thread */
cb[2].aio_fildes   = fd;
cb[2].aio_offset   = 0;
cb[2].aio_buf      = buffer [2];
cb[2].aio_nbytes   = 256;
cb[2].aio_reqprio  = 0;
cb[2].aio_sigevent.sigev_notify = SIGEV_THREAD;
cb[2].aio_sigevent.sigev_notify_function   = thread;
cb[2].aio_sigevent.sigev_notify_attributes = NULL;
cb[2].aio_sigevent.sigev_value.sival_ptr = & cb[2];

/* Lancement des lectures */
if ((aio_read(& cb[0]) < 0)
```

```
      || (aio_read(& cb[1]) < 0)
      || (aio_read(& cb[2]) < 0)) {
         perror("aio_read");
         exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Lectures lancées \n");
    while ((aio_error(& cb[0]) == EINPROGRESS)
        || (aio_error(& cb[1]) == EINPROGRESS)
        || (aio_error(& cb[2]) == EINPROGRESS))
        sleep(1);
    nb_octets = aio_return(& cb[0]);
    fprintf(stdout, "Lecture 0 : %d octets lus \n", nb_octets);
    return EXIT_SUCCESS;
}
```

La vérification `(aio_error(cb) == EINPROGRESS)` est indispensable dans le gestionnaire de signaux, car `SIGRTMIN+3` peut provenir d'une autre source. Au sein du thread ce contrôle est inutile car on ne doit normalement pas appeler cette routine directement. Je l'ai laissée car c'est une bonne habitude – paranoïaque – pour s'assurer de la fin d'un transfert avant d'appeler `aio_return()`.

En attendant que toutes les lectures soient terminées, le programme s'endort par période d'une seconde – ou moins quand le signal arrive – pour éviter de consommer inutilement de ressources CPU. L'exécution se déroule comme prévu :

```
$ ./exemple_aio_read exemple_aio_read
Lectures lancées
Lecture 1 : 256 octets lus
Lecture 2 : 256 octets lus
Lecture 0 : 256 octets lus
$ ls -l Makefile
-rw-r--r--   1 ccb      ccb       242 Mar  2 14:02 Makefile
$ ./exemple_aio_read Makefile
Lectures lancées
Lecture 1 : 242 octets lus
Lecture 2 : 242 octets lus
Lecture 0 : 242 octets lus
$
```

Étant donné qu'une lecture ou une écriture asynchrone modifie la position du pointeur dans le fichier, il faut considérer que cette valeur peut changer à tout moment tant que l'opération n'est pas terminée. Et à ce moment encore la position restera indéterminée tant qu'elle n'aura pas été revalidée avec `lseek()`.

Cela signifie qu'il faut absolument éviter d'employer de lecture ou d'écriture synchrones habituelles pendant que des opérations asynchrones ont lieu. Il faudrait en effet, dans le cours normal du processus, lier atomiquement le déplacement du pointeur avec `lseek()` et l'appel-système `read()` ou `write()` qui suit. Ceci ne peut se faire qu'à l'aide des appels-système `pread()` et `pwrite()`[1], que nous avons examinés dans le chapitre 19 et qui sont employés par la bibliothèque C pour implémenter `aio_read()` et `aio_write()`.

1. Ces appels-système sont apparus dans Linux 2.2, ce qui explique pourquoi les entrées-sorties asynchrones n'étaient pas disponibles auparavant, même si ce ne sont que des fonctions de bibliothèque.

Lorsque plusieurs opérations simultanées doivent être accomplies sur le même fichier ou sur des fichiers différents, il est possible de programmer un ensemble d'entrées-sorties avec **lio_listio()** :

```
int lio_listio (int mode, struct aiocb * liste_aiocb [], int nb_aiocb,
                struct sigevent * notification);
```

Le premier argument est le mode de fonctionnement de `lio_listio()`. Il peut prendre l'une des deux valeurs suivantes :

- **LIO_NOWAIT** : la fonction lance toutes les opérations décrites dans les arguments suivants de manière asynchrone et se termine. Une fois que toutes les opérations auront été réalisées, le processus recevra une notification décrite dans le dernier argument de `lio_listio()`. Naturellement, les notifications individuelles sont également reçues au fur et à mesure de l'accomplissement des travaux.
- **LIO_WAIT** : la fonction attend pour se terminer que toutes les opérations soient finies. Ce mécanisme est surtout utilisé avec une seule opération à la fois, pour réaliser une lecture ou une écriture normale, synchrone, alors que des opérations asynchrones ont lieu sur le même fichier. Le système préserve en effet l'atomicité du positionnement du pointeur et de l'entrée-sortie sur le fichier.

Le second argument est un tableau de pointeurs sur des structures `aiocb`. Il y a donc un niveau d'indirection supplémentaire. L'avantage c'est qu'un pointeur `NULL` dans ce tableau est ignoré. On peut donc préparer une table avec de nombreuses opérations et remplir le tableau de pointeurs en ignorant facilement des opérations qu'on ne souhaite pas effectuer immédiatement. Le troisième argument est le nombre d'opérations dans le tableau.

Dans chaque structure `aiocb`, il faut à présent remplir le champ `aio_lio_opcode` avec l'une des valeurs suivantes :

- **LIO_READ** : on veut faire une lecture.
- **LIO_WRITE** : on veut une écriture.
- **LIO_NOP** : pour ignorer l'opération.

Pour utiliser `lio_listio()` au lieu du lancement successif des trois lectures asynchrones de l'exemple précédent, on ajoute dans les variables de `main()` la structure `sigevent` et la table

```
    struct sigevent  lio_sigev;
    struct aiocb *   lio [3];
```

puis on lance les lectures ainsi :

```
/* Lancement des lectures */
    lio[0] = & cb[0];
    lio[1] = & cb[1];
    lio[2] = & cb[2];
    lio_sigev.sigev_notify = SIGEV_NONE;
    if (lio_listio(LIO_NOWAIT, lio, 3, & lio_sigev) < 0) {
        perror("lio_listio");
        exit(EXIT_FAILURE);
    }
```

Le déroulement du programme est identique à celui de `exemple_aio_read` :

```
$ ./exemple_lio_listio exemple_lio_listio
Lectures lancées
Lecture 1 : 256 octets lus
Lecture 2 : 256 octets lus
Lecture 0 : 256 octets lus
$
```

Dans notre programme nous avons laissé le processus dans de courtes périodes de sommeil entre lesquelles nous avons examiné l'état des opérations en cours. Il existe une fonction plus adaptée à cette attente, nommée **`aio_suspend()`** :

```
int aio_suspend (const struct aiocb * liste_aiocb [], int nb_aiocb,
                 const struct timespec * delai_maxi);
```

Cette routine prend en argument un tableau de pointeurs sur des structures `aiocb`, comme le faisait `lio_listio()`, et attend que l'une au moins des opérations du tableau se termine. Elle rend alors la main au processus. On peut ensuite examiner, avec `aio_error()`, quelle opération s'est achevée et vérifier son code de retour avec `aio_return()`. L'opération terminée peut être supprimée de la liste d'attente en remplaçant son pointeur par `NULL`.

On peut ainsi éviter les notifications par l'intermédiaire d'un gestionnaire de signaux impliquant un changement de contexte du processus. Le programme ci-dessous emploie ce principe et n'utilise ni gestionnaire de signaux ni thread supplémentaire.

Si `aio_suspend()` est interrompue par un signal, elle échoue avec l'erreur `EINTR`. Ce cas peut être tout à fait normal s'il s'agit du signal notifiant la fin d'une opération. Le dernier argument est un délai d'attente maximal. Si rien ne s'est produit durant ce temps, la fonction échoue avec l'erreur `EAGAIN`. Dans le programme suivant, nous n'employons pas de délai, aussi ce pointeur est-il `NULL`.

exemple_aio_suspend.c :

```
    #include <aio.h>
    #include <errno.h>
    #include <stdio.h>
    #include <sys/stat.h>

    #define NB_OP   10

    int
main (int argc, char * argv[])
{
    int           fd;
    int           i;
    struct aiocb  cb[NB_OP];
    char          buffer[256][NB_OP];

    struct sigevent  lio_sigev;
    struct aiocb *   lio[NB_OP];

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier \n", argv[0]);
        exit(EXIT_FAILURE);
```

```
    }
    if ((fd = open(argv[1], O_RDONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < NB_OP; i ++) {
        cb[i].aio_fildes   = fd;
        cb[i].aio_offset   = 0;
        cb[i].aio_buf      = buffer[i];
        cb[i].aio_nbytes   = 256;
        cb[i].aio_reqprio  = i;
        cb[i].aio_lio_opcode = LIO_READ;
        cb[i].aio_sigevent . sigev_notify = SIGEV_NONE;
        lio[i]= & cb[i];
    }

    lio_sigev.sigev_notify = SIGEV_NONE;
    if (lio_listio(LIO_NOWAIT, lio, NB_OP, & lio_sigev) < 0) {
        perror("lio_listio");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Lectures lancées \n");

    while (1) {
        /* Reste-t-il des opérations en cours */
        for (i = 0; i < NB_OP; i ++)
            if (lio[i] != NULL)
                break;
        if (i == NB_OP)
            /* Toutes les opérations sont finies */
            break;
        if (aio_suspend(lio, NB_OP, NULL) == 0) {
            for (i = 0; i < NB_OP; i ++)
                if (lio[i] != NULL)
                    if (aio_error(lio[i]) != EINPROGRESS) {
                        fprintf(stdout, "Lecture %d : %d octets \n",
                                        i, aio_return(lio[i]));
                        /* fini... */
                        lio[i] = NULL;
                }
        }
    }
    return EXIT_SUCCESS;
}
```

Nous lançons dix lectures simultanées ; nous obtenons :

```
$ ./exemple_aio_suspend exemple_aio_read
Lectures lancées
Lecture 0 : 256 octets
Lecture 1 : 256 octets
Lecture 2 : 256 octets
Lecture 3 : 256 octets
```

```
Lecture 4 : 256 octets
Lecture 5 : 256 octets
Lecture 6 : 256 octets
Lecture 7 : 256 octets
Lecture 8 : 256 octets
Lecture 9 : 256 octets
$
```

Mentionnons enfin l'existence d'une fonction **aio_cancel()** permettant théoriquement d'annuler une opération qui n'a pas encore eu lieu.

```
int aio_cancel (int fd, struct aiocb * aiocb);
```

Cette routine tente d'annuler l'opération indiquée en second argument sur le descripteur de fichier fourni en première position. Si le pointeur `aiocb` est `NULL`, cette fonction tente d'annuler toutes les opérations ayant lieu sur le descripteur indiqué.

La routine `aio_cancel()` ne donnant aucune garantie de réussite et ne permettant pas de savoir si l'opération a réellement été annulée, elle n'a quasiment aucune utilité.

Écritures synchronisées

Les mécanismes d'entrée-sortie avancés, tels que ceux qui sont décrits par la norme SUSv3, introduisent un concept d'écritures synchronisées, qui ne doivent pas être confondues avec les écritures synchrones ou asynchrones. En fait, il est tout à fait possible d'employer des écritures asynchrones synchronisées.

La notion d'écriture synchronisée fait référence au transfert effectif des données vers le disque. Nous avons déjà observé dans le chapitre 18 que des informations écrites dans un flux traversaient trois niveaux de buffers successifs (voir figure 18.1). Ici, nous travaillons directement avec le descripteur de fichier et nous ignorons donc le premier buffer associé au flux[1]. Nous ne pouvons pas contrôler non plus la zone tampon intégrée dans le lecteur de disque, mais nous allons nous intéresser à la mémoire cache gérée par le noyau.

Il peut être utile dans certaines applications temps-réel, dans des systèmes de gestion de bases de données ou dans des logiciels d'enregistrement de type « boîte noire », de pouvoir passer outre la mémoire cache du noyau et s'assurer que les données écrites par un appel-système `write()` ont bien été transmises au contrôleur de périphérique, à défaut du support physique réel. Ceci peut être réalisé avec plusieurs degrés de précision.

Tout d'abord rappelons que l'appel-système `fsync()`, décrit dans le chapitre 19, sert à synchroniser le contenu du fichier sur lequel on lui passe un descripteur. Cette routine attend que toutes les données écrites soient effectivement transmises au contrôleur de disque, puis elle revient en renvoyant zéro si tout s'est bien passé, ou –1 sinon. Dans le cas d'un échec de synchronisation, l'erreur **EIO** est renvoyée dans `errno`.

Il existe également un appel-système `sync()` qui ne prend pas d'argument et renvoie toujours zéro. Ici, toutes les écritures en attente dans la mémoire cache du noyau sont réalisées avant le retour. Comme nous l'avons précisé dans le chapitre 18, il existe un utilitaire `/bin/sync` qui

1. Les méthodes d'écritures synchronisées étudiées ici sont appliquées la plupart du temps directement aux descripteurs pour des raisons d'efficacité. Si toutefois l'utilisation d'un flux est indispensable, on pourra se tourner vers `fflush()` ou `setvbuf()`, que nous avons rencontrées dans le chapitre 18.

invoque cet appel-système. Sur les premières versions de Linux, les écritures pouvaient rester en suspens indéfiniment tant que la mémoire cache n'était pas pleine et qu'on n'invoquait pas `sync()`. Pour cela, l'utilitaire `/bin/sync` était appelé régulièrement par une commande de la table `crond`. À présent, ce n'est plus nécessaire, un démon particulier nommé `kflushd` est chargé de ce rôle. Il est créé directement par le noyau au moment du démarrage (juste après le lancement de *init*).

L'utilisation de `fsync()` peut parfois être suffisante dans une application, car elle permet de créer des points où on connaît l'état du fichier. Le vidage de la mémoire tampon est quand même assez coûteux, d'autant qu'il faut mettre à jour non seulement les données proprement dites, mais également des informations de contrôle qui ne sont pas nécessairement indispensables, comme la date de modification de l'i-nœud. Pour cela, Posix.1b a introduit l'appel-système **`fdatasync()`**, disponible si la constante **`_POSIX_SYNCHRONIZED_IO`** est définie dans `<unistd.h>` :

```
int fdatasync (int descripteur);
```

Cette routine se comporte comme `fsync()` au niveau de l'application, mais elle n'écrit réellement que les données indispensables, en laissant les autres dans la mémoire cache du noyau. En cas d'arrêt brutal du système, l'état du disque est tel que les informations pourront être récupérées, éventuellement après le passage d'un utilitaire de réparation comme `/sbin/fsck.ext3`.

Si toutes les écritures dans un fichier doivent être considérées de la même manière, il est agaçant de devoir invoquer `fsync()` ou `fdatasync()` après chaque `write()`. Pour éviter cette manipulation, nous pouvons configurer directement le comportement de toutes les écritures grâce à l'appel-système `open()`. L'attribut `O_SYNC` ajouté lors de l'ouverture d'un descripteur signifie que toutes les écritures seront synchronisées, comme si on invoquait `fsync()` immédiatement après.

> **Note**
> Le noyau Linux n'autorise pas la modification de l'attribut `O_SYNC` d'un fichier après son ouverture, contrairement à d'autres systèmes qui le permettent avec `fcntl()`. Ceci peut d'ailleurs poser des problèmes lorsque le descripteur a été hérité du processus père.

Il existe deux autres constantes différentes tolérées lors de l'ouverture d'un descripteur, bien qu'elles aient pour l'instant exactement la même signification que `O_SYNC` :

Nom	Signification
`O_DSYNC`	Pour ne synchroniser automatiquement que les données écrites dans le descripteur, sans se soucier des informations relatives à l'i-nœud. C'est l'équivalent d'une `fdatasync()` après chaque `write()`.
`O_RSYNC`	Lors d'un `read()`, le noyau doit mettre à jour l'heure de dernière lecture de l'i-nœud. Avec cette constante, cette mise à jour sera synchronisée. Lorsque `read()` se termine, l'i-nœud a été mis à jour. Ce mécanisme est rarement utile.

Comme nous l'avons déjà indiqué, les écritures synchronisées peuvent également se faire de manière asynchrone. Lorsque la notification sera envoyée au processus, les données écrites

auront été entièrement transférées sous la houlette du contrôleur de périphérique. Cela se fait en utilisant `O_SYNC` lors de l'ouverture du descripteur.

Pour obtenir l'équivalent asynchrone des fonctions `fsync()` et `fdatasync()`, c'est-à-dire la garantie ponctuelle de vidage de la mémoire cache, on emploie la fonction **`aio_fsync()`**, déclarée ainsi :

```
int aio_fsync (int mode, struct aiocb * aiocb);
```

Cette routine déclenche `fsync()` – de manière asynchrone – sur le descripteur `aiocb.aio_fildes` si son premier argument vaut `O_SYNC`, ou simplement `fdatasync()` s'il s'agit de `O_DSYNC`. Lorsque la synchronisation est terminée, la notification inscrite dans `aiocb.aio_sigevent` est déclenchée. Il faut bien comprendre que la routine `aio_fsync()` n'attend pas la fin du vidage de la mémoire cache. Si on désire obtenir ce comportement, il faut appeler directement `fsync()`.

Les écritures synchronisées sont évidemment très coûteuses en temps d'exécution. Le programme suivant va en faire la démonstration : il crée un fichier dans lequel il écrit 256 × 1 024 blocs de 256 octets. Suivant la valeur du second argument sur la ligne de commande, les écritures seront synchronisées ou pas.

exemple_osync.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <unistd.h>

    int
main (int argc, char * argv [])
{
    int fd;
    char buffer[256];
    int i, j;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s fichier sync \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((argv[2][0] == 'o') || (argv[2][0] == 'O')) {
        fprintf(stdout, "Écritures synchronisées \n");
        if ((fd = open(argv[1], O_RDWR | O_CREAT | O_SYNC, 0644)) < 0) {
            perror("open");
            exit(EXIT_FAILURE);
        }
    } else {
        fprintf(stdout, "Écritures non synchronisées \n");
        if ((fd = open(argv[1], O_RDWR | O_CREAT, 0644)) < 0) {
            perror("open");
            exit(EXIT_FAILURE);
        }
    }
    for (i = 0; i < 1024; i ++)
        for (j = 0; j < 256; j ++)
            if (write(fd, buffer, 256) < 0) {
```

```
                perror("write");
                exit(EXIT_FAILURE);
            }
    fsync(fd);
    close(fd);
    return EXIT_SUCCESS;
}
```

L'appel `fsync()` final nous permet d'être sûr que toutes les écritures ont eu lieu au moment de la fin du programme. Sinon des transferts continueraient à se produire alors que nous serions déjà revenus au shell, faussant ainsi les résultats.

Pour avoir des statistiques d'exécution assez précises, on pourrait utiliser la commande `time` du shell et invoquer le programme plusieurs fois afin d'obtenir des valeurs moyennes. Les différences sont telles qu'il n'y a même pas besoin d'utiliser une surveillance si précise. Il suffit d'encadrer l'appel au programme par des commandes `date` pour connaître sa durée :

```
$ date   ;     ./exemple_osync essai.sync O   ;     date
lun mar  6 14:09:54 CET 2000
Écritures synchronisées
lun mar  6 14:23:26 CET 2000
$ date   ;     ./exemple_osync essai.sync N    ;    date
lun mar  6 14:23:30 CET 2000
Écritures non synchronisées
lun mar  6 14:23:54 CET 2000
$ rm essai.sync
$
```

24 secondes dans un cas, contre près de 14 minutes dans l'autre !

La différence est aussi importante car nous avons demandé de nombreuses écritures successives de blocs de petite taille. L'i-nœud du fichier doit donc être mis à jour pour chaque écriture (taille du fichier, heure de dernière modification…). Lors d'écritures non synchronisées, cet i-nœud reste en mémoire entre chaque modification. Il n'est écrit sur le disque qu'à une ou deux reprises – à cause du démon `pdflush` entre autres. De même, chaque bloc disque (1 Ko) est touché successivement par quatre écritures de 256 octets, et on gagne largement à le garder en mémoire le plus longtemps possible.

On restreindra donc l'utilisation des écritures synchronisées aux applications qui en ont réellement besoin, avec des contraintes importantes en tolérance de panne. Dans la plupart des cas, un simple appel à `fsync()` en des points-clés du logiciel suffira pour les besoins de fiabilité, tout en conservant un bon temps de réponse à l'application. Rappelons que l'écriture synchronisée garantit uniquement que les données sont parvenues au contrôleur de disque, mais pas qu'elles sont effectivement écrites sur le support physique. Si ce point devient critique – comme dans un système d'enregistrement embarqué –, il faudra choisir avec soin le matériel utilisé afin de minimiser la latence des écritures effectives.

Conclusion

Nous avons examiné ici plusieurs méthodes permettant d'améliorer les entrées-sorties d'un processus, dans le but de rendre les écritures plus fiables (synchronisées) ou de rendre le fil d'exécution principal du programme indépendant des événements survenant sur les descripteurs (multiplexage et entrées-sorties asynchrones).

Les mécanismes de multiplexage sont bien entendu applicables sur les tubes de communication, mais ils sont le plus fréquemment utilisés avec les sockets, qui sont une extension de ces tubes à l'échelle d'un réseau. Les prochains chapitres vont développer les concepts et les principes mis en œuvre dans ce type de logiciel.

31

Programmation réseau

Nous allons essayer dans ce chapitre de mettre en place les bases de la programmation réseau sous Linux, principalement en ce qui concerne la détermination des adresses et des numéros de ports, ainsi que la manipulation de l'ensemble de ces données.

L'essentiel du travail dans la programmation réseau revient en effet à déterminer comment joindre le correspondant. La communication elle-même ne diffère pas beaucoup des méthodes observées dans le chapitre 28 avec les tubes. Ceci sera abordé dans le chapitre suivant, par l'intermédiaire de l'interface proposée par les sockets BSD.

Réseaux et couches de communication

Le but de notre étude est de permettre la mise au point d'applications pouvant recevoir ou envoyer des informations, en dialoguant avec des correspondants se trouvant n'importe où dans le monde, à partir du moment où une connectivité réseau a été établie.

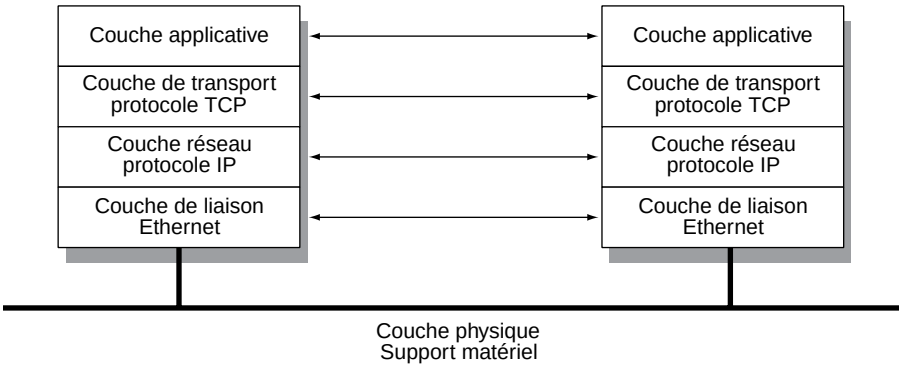
On représente communément les fonctionnalités réseau par une série de couches successives de communication. Ce modèle est intéressant car il permet de bien distinguer la manière dont les différents protocoles sont liés. Chaque couche ne peut dialoguer directement qu'avec la couche supérieure ou inférieure, seule la couche physique peut mettre en relation les différentes stations. Un exemple de stratification réseau est présenté dans la figure 31.1.

Lorsqu'une application désire envoyer des informations à une autre application se déroulant sur un ordinateur distant, elle prépare un paquet de données qu'elle transmet à la couche de transport (TCP sur ce schéma). Celle-ci encadre les données avec ses propres informations – en l'occurrence des champs servant à s'assurer de l'intégrité du message transmis – puis passe le paquet à la couche réseau, IP. Cette dernière encadre à nouveau le paquet par des informations permettant le routage dans le réseau et passe le relais à la couche de liaison, Ethernet. À ce niveau, les dernières informations ajoutées permettent d'identifier la carte réseau de l'ordinateur cible. Le niveau matériel assure la transmission électrique des données. À l'arrivée, le processus inverse se déroule, chaque couche supprime les éléments qui lui

étaient propres et donne le paquet restant à la couche supérieure. Finalement, la couche applicative reçoit les données qui lui étaient destinées. On peut donc considérer que chaque couche dialogue virtuellement avec la couche correspondante sur l'ordinateur cible, bien qu'elle n'ait de véritable contact qu'avec ses couches supérieure et inférieure.

Les noms des différentes couches sont dérivés d'un document de 1984 nommé modèle OSI (*Open Systems Interconnection*), qui sert à représenter les communications réseau avec sept niveaux successifs. Malheureusement les protocoles les plus répandus, TCP/IP et UDP/IP, ne sont pas fondés sur ce modèle et n'emploient que cinq niveaux, comme on le voit sur la figure. Les termes ont été conservés par habitude, mais ils ne conviennent pas tout à fait.

Figure 31.1
Exemple de couches de communication



Le support physique permettant de relier des stations peut prendre des formes diverses. Les plus communes sont les interfaces Ethernet, avec une liaison en câble fin (prise BNC) ou en paires torsadées (prise RJ45), et les liaisons modems. Entre deux stations données peuvent se trouver de nombreux éléments, comme des répéteurs qui assurent la prolongation d'un brin physique, des passerelles qui permettent l'interconnexion de réseaux différents, ou des routeurs qui servent à orienter les données entre plusieurs sous-réseaux. Sur un réseau, les machines sont identifiées de manière unique. Les cartes Ethernet – qui composent la couche de liaison – comportent par exemple un identificateur numérique sur 48 bits, appelé adresse MAC (*Medium Access Control*), dont l'unicité est assurée par le fabricant de la carte. Cette valeur peut être examinée à l'aide de l'utilitaire `/sbin/ifconfig` par exemple :

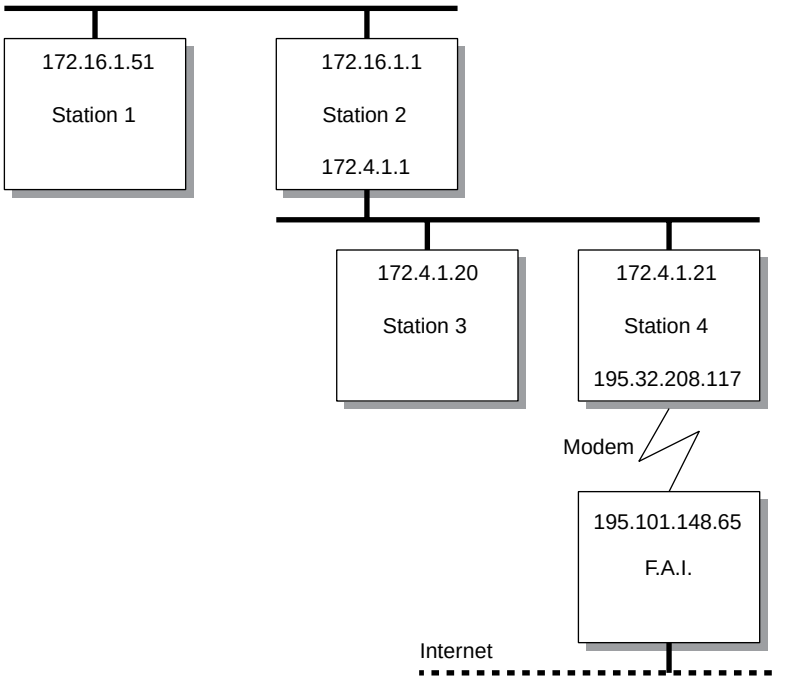
```
$ /sbin/ifconfig eth0
eth0      Lien encap:Ethernet  HWaddr 00:50:04:8C:82:5E
          inet adr:172.16.1.51  Bcast:172.16.1.255  Masque:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Paquets Reçus:117 erreurs:0 jetés:0 débordements:0 trames:0
          Paquets transmis:66 erreurs:0 jetés:0 débordements:0 carrier:0
          collisions:0 lg file transmission:100
          Interruption:3 Adresse de base:0x200
$
```

L'identificateur MAC (`00:50:04:8C:82:5E` en l'occurrence) est indiqué dans la rubrique *Hardware Address*. Une machine est capable, grâce à son adresse, de reconnaître si un bloc de données lui est destiné et de le transmettre aux protocoles de dialogue se trouvant au-dessus. On trouvera au besoin plus de renseignements dans [FERRERO 1993] *Les réseaux Ethernet*.

Plusieurs protocoles peuvent être employés pour transmettre des informations au-dessus de cette couche de liaison, mais l'essentiel des communications au niveau applicatif se fait en employant IP (*Internet Protocol*). Le protocole IP permet d'envoyer un paquet de données à destination d'un hôte particulier, en l'identifiant à l'aide d'une adresse sur 4 octets. Celle-ci est presque toujours représentée avec la notation dite « pointée », c'est-à-dire en écrivant les valeurs décimales des octets séparées par des points. Dans l'exemple précédent, `/sbin/ifconfig` affichait l'adresse IP de l'interface `eth0` (`172.16.1.51`) avec le titre *Internet Address*. Le protocole actuel IP version 4 sera remplacé dans l'avenir par IP version 6 (il n'existe pas de version 5), aussi appelé IPng (*IP Next Generation*), mais le support dans le noyau Linux est encore expérimental et incomplet.

Au niveau du protocole IP, les données peuvent être routées. La communication n'est plus limitée aux machines se trouvant sur le même réseau matériel[1]. Au contraire, il existe des passerelles permettant de transférer les paquets d'un réseau vers un autre. La figure 31.2 montre un exemple de réseaux reliés entre eux. Chaque machine peut dialoguer avec toutes les autres, par le biais de la couche IP.

Figure 31.2
Exemple de passerelles entre réseaux



Par exemple le noyau de la station 1 sait, grâce à ses tables de routage configurées avec l'utilitaire `/sbin/route`, que pour atteindre une machine ne se trouvant pas directement sur son brin Ethernet, par exemple la station 3, il doit demander le relais à la station 2. Les paquets transmis à la couche de liaison seront donc dirigés vers l'adresse MAC de cette machine 2.

1. La mise en correspondance entre l'adresse IP et l'adresse MAC se fait par l'intermédiaire d'un protocole nommé ARP (*Address Resolution Protocol*) sortant du cadre de notre propos.

Par contre, l'adresse IP du destinataire sera celle de la station 3. La couche de liaison s'occupe uniquement de l'adresse Ethernet et pas de l'adresse IP.

La station 2 dispose de deux cartes Ethernet. Son noyau est configuré grâce à `/sbin/ipfwadm` pour laisser passer les paquets d'un réseau à l'autre. Lorsqu'un bloc de données arrive sur une carte réseau à destination d'un autre sous-réseau, celle-ci assure le transfert d'une interface à l'autre.

La station 3 sait que pour accéder aux machines dont l'adresse IP commence par `172.16`, elle doit s'adresser également à la station 2. Quant à la station 4, elle sert à joindre un fournisseur d'accès Internet. Elle sert aussi de passerelle, mais comme les machines des réseaux `172.16` et `172.4` ne sont pas connues directement sur Internet, la configuration est légèrement plus compliquée car il faut employer un mécanisme de *Masquerading IP*. On trouvera des renseignements sur toutes ces notions de routage dans [KIRCH 2001] *L'administration réseau sous Linux*, et dans les documents *NET-3-HOWTO et IP-Masquerade mini-HOWTO*.

Lorsqu'on utilise une connexion avec le protocole PPP (*Point to Point Protocol*), comme c'est le cas avec la majorité des fournisseurs d'accès à Internet, il n'y a pas vraiment de distinction entre la couche réseau et la couche de liaison, qui sont regroupées dans PPP.

Ce qu'on retiendra ici, c'est que le protocole IP est capable d'envoyer un paquet de données à destination d'un hôte précis, dont l'adresse est indiquée par 4 octets, en franchissant les éléments de routage.

L'utilisation directe du protocole IP est plutôt rare au niveau d'une application. On peut l'employer pour envoyer des messages de commande appartenant au protocole ICMP (*Internet Control Message Protocol*), comme les demandes d'écho émises par l'utilitaire `ping`. Néanmoins, la plupart du temps on fera appel à une couche supérieure. Nous étudierons ici les deux protocoles TCP (*Transmission Control Protocol*) et UDP (*User Datagram Protocol*), qui ont des rôles complémentaires.

Le protocole TCP sert à fiabiliser la communication entre deux hôtes. Pour cela, il assure les fonctionnalités suivantes :

- Connexion. Avec ce protocole, une liaison s'établit par une concertation de l'émetteur et du récepteur. On dit que la communication s'effectue de manière connectée. Une fois le canal de communication établi, il reste en vigueur jusqu'à ce qu'on le referme.
- Fiabilité. Le protocole TCP garantit que – tant que la connexion sera valide – les données qui y transitent arriveront dans l'ordre et que leur intégrité sera vérifiée.
- Contrôle de flux. En complément de la fiabilité du protocole TCP, il est possible de l'employer comme un flux d'octets, à la manière d'un tube de communication. L'écriture peut devenir bloquante si le récepteur ne lit pas suffisamment vite de son côté.

À l'opposé, le protocole UDP fournit un service de transmission de paquets (*datagram*) sans assurer de fiabilité :

- Pas de connexion. L'émetteur peut envoyer des données sans s'assurer qu'un processus est à l'écoute. Aucun acquittement n'est nécessaire.
- Pas de fiabilité. Le paquet transmis ne contient qu'optionnellement une somme de contrôle. Si des données sont perdues ou erronées, elles ne sont pas répétées.
- Transmission en paquets. Les données envoyées n'arrivent pas nécessairement dans le même ordre qu'au départ.

En fait, le protocole IP brut offre les mêmes fonctionnalités que UDP, à la somme de contrôle près, qui n'existe pas dans la couche réseau (en fait IP vérifie l'intégrité de ses propres informations mais pas celles des données du paquet). L'implémentation de TCP effectue donc de nombreuses tâches afin d'assurer un mécanisme de communication fiable. Entre autres, TCP vérifie l'état des paquets qui arrivent, s'assure qu'ils sont dans le bon ordre au moyen d'un numéro de séquence, gère un délai maximal de transmission, des acquittements, etc. Si un paquet est endommagé ou absent, la couche TCP du destinataire demande à la couche TCP de l'émetteur de renvoyer les données. Tout ceci offre donc une sécurité de transmission des informations mais au prix d'une charge réseau supplémentaire.

Le protocole UDP de son côté permet d'envoyer des paquets de données sans se soucier vraiment du récepteur. Ceci est particulièrement utile dans les applications qui veulent diffuser des informations sous forme de fonctionnalité annexe du logiciel. Il n'est pas question dans ce cas de perdre du temps à gérer les connexions des correspondants ni de risquer de rester bloqué si le récepteur ne lit pas assez vite. L'émetteur peut envoyer ses paquets de données et passer immédiatement à autre chose, il est de la responsabilité du récepteur d'être à l'écoute au bon moment.

Protocoles

Les protocoles connus par le système dépendent des options de compilation du noyau. Toutefois, un certain nombre d'entre eux sont définis dans un fichier système nommé `/etc/protocols`.

```
$ cat /etc/protocols
# /etc/protocols:
ip        0    IP          # internet protocol, pseudo protocol number
icmp      1    ICMP        # internet control message protocol
igmp      2    IGMP        # Internet Group Management
ggp       3    GGP         # gateway-gateway protocol
ipencap   4    IP-ENCAP    # IP encapsulated in IP (officially ``IP'')
st        5    ST          # ST datagram mode
tcp       6    TCP         # transmission control protocol
egp       8    EGP         # exterior gateway protocol
pup       12   PUP         # PARC universal packet protocol
udp       17   UDP         # user datagram protocol
hmp       20   HMP         # host monitoring protocol
xns-idp   22   XNS-IDP     # Xerox NS IDP
rdp       27   RDP         # "reliable datagram" protocol
iso-tp4   29   ISO-TP4     # ISO Transport Protocol class 4
xtp       36   XTP         # Xpress Tranfer Protocol
ddp       37   DDP         # Datagram Delivery Protocol
idpr-cmtp 39   IDPR-CMTP   # IDPR Control Message Transport
rspf      73   RSPF        #Radio Shortest Path First.
vmtp      81   VMTP        # Versatile Message Transport
ospf      89   OSPFIGP     # Open Shortest Path First IGP
ipip      94   IPIP        # Yet Another IP encapsulation
encap     98   ENCAP       # Yet Another IP encapsulation
$
```

À chaque protocole est associé un numéro d'identification standard, employé pour la communication réseau. Ces numéros ne nous intéressent pas ici. L'important pour nous est de connaître l'orthographe connue par le système pour les noms des protocoles qui nous concernent, c'est-à-dire UDP, TCP, IP, éventuellement RDP et ICMP.

Pour analyser ce fichier, la bibliothèque met à notre disposition plusieurs fonctions. Tout d'abord **getprotobyname()** permet de rechercher un protocole à partir d'une chaîne représentant son nom, alors que **getprotobynumber()** effectue le même travail à partir du numéro du protocole. Ces fonctions sont déclarées dans `<netdb.h>`. Ce fichier d'en-tête contient une déclaration qui déclenche un avertissement du compilateur si on laisse l'option `–pedantic`. On peut ignorer cet avertissement ou supprimer cette option dans le fichier `Makefile`.

```
struct protoent * getprotobyname   (const char * nom);
struct protoent * getprotobynumber (int numero);
```

La structure **protoent** contient les membres suivants :

Nom	Type	Signification
p_name	char *	Nom officiel du protocole (défini par la RFC 1700).
p_proto	int	Numéro officiel du protocole (dans l'ordre des octets de la machine).
p_aliases	char **	Table de chaînes de caractères correspondant à d'éventuels alias. Cette table est terminée par un pointeur NULL.

Nous avons indiqué que le numéro de protocole est fourni dans l'ordre des octets de la machine. Nous détaillerons ceci plus loin.

L'utilisation de ces routines est assez évidente. Le programme suivant affiche les informations concernant les protocoles indiqués sur la ligne de commande.

exemple_getprotoby.c :

```
    #include <netdb.h>
    #include <stdio.h>

    int
main (int argc, char * argv[])
{
    int               i, j;
    int               numero;
    struct protoent * proto;

    for (i = 1; i < argc; i ++) {
        if (sscanf(argv[i], "%d", & numero) == 1)
            proto = getprotobynumber(numero);
        else
            proto = getprotobyname(argv[i]);
        fprintf(stdout, "%s : ", argv[i]);
        if (proto == NULL) {
            fprintf(stdout, "inconnu \n");
            continue;
        }
```

```
        fprintf(stdout, "%s ( ", proto->p_name);
        for (j = 0; proto->p_aliases[j] != NULL; j ++)
            fprintf(stdout, "%s ", proto->p_aliases[j]);
        fprintf(stdout, ") numéro = %d \n", proto->p_proto);
    }
    return EXIT_SUCCESS;
}
```

Nous pouvons rechercher quelques protocoles, en vérifiant que la distinction entre majuscules et minuscules se fait.

```
$ ./exemple_getprotoby tcp 1
tcp : tcp ( TCP ) numéro = 6
1 : icmp ( ICMP ) numéro = 1
$ ./exemple_getprotoby udp 17 UDP UdP
udp : udp ( UDP ) numéro = 17
17 : udp ( UDP ) numéro = 17
UDP : udp ( UDP ) numéro = 17
UdP : inconnu
$
```

Si on désire balayer tous les protocoles connus – par exemple pour comparer les noms avec `strcasecmp()` afin d'autoriser des saisies comme UdP –, on peut employer les routines **`setprotoent()`**, **`getprotoent()`** et **`endprotoent()`**. La première ouvre le fichier des protocoles, la seconde y lit l'enregistrement suivant, et la dernière referme ce fichier.

```
void              setprotoent (int ouvert);
struct protoent * getprotoent (void);
void              endprotoent (void);
```

Si l'argument passé à `setprotoent()` n'est pas nul, les appels éventuels à `getprotobyname()` ou `getprotobynumber()` ne refermeront pas le fichier après l'avoir consulté. Sinon, la lecture reprendra au début du fichier.

Le programme suivant affiche le nom de tous les protocoles connus par le système :

exemple_getprotoent.c :

```
    #include <stdio.h>
    #include <netdb.h>

    int
main (void)
{
    struct protoent * proto;
    setprotoent(0);
    while ((proto = getprotoent()) != NULL)
        fprintf(stdout, "%s ", proto->p_name);
    endprotoent();
    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

L'exécution donne :

```
$ ./exemple_getprotoent
ip icmp igmp ggp ipencap st tcp egp pup udp hmp xns-idp rdp iso-tp4 xtp ddp
➥ idpr-cmtp rspf vmtp ospf ipip encap
$
```

Les routines que nous avons vues ici renvoient leurs données dans des zones de mémoire statiques. Ceci peut poser un problème dans un programme multithread, aussi existe-t-il des extensions Gnu réentrantes.

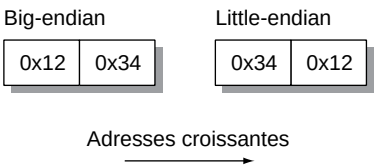
```
int getprotobynumber_r (int numero,
                        struct protoent * protocole,
                        char * buffer, size_t taille_buffer,
                        struct protoent ** retour);
int getprotobyname_r   (const char * nom,
                        struct protoent * protocole,
                        char * buffer, size_t taille_buffer,
                        struct protoent ** retour);
int getprotoent_r      (struct protoent * protocole,
                        char * buffer, size_t taille_buffer,
                        struct protoent ** retour);
```

Ces routines sont un peu plus compliquées puisqu'il faut leur transmettre un buffer dans lequel elles inscriront les chaînes de caractères sur lesquelles la structure `protoent` contient des pointeurs. Nous avons déjà rencontré ce principe dans le chapitre 26, avec diverses routines comme `getgrnam_r()`.

Ordre des octets

Les communications et les échanges de données entre ordinateurs hétérogènes sont souvent confrontés au problème d'ordre des octets dans les valeurs entières. Pour stocker en mémoire une valeur tenant sur 2 octets, certains processeurs placent en première position l'octet de poids faible, puis celui de poids fort. Comme les données commencent par leur plus petite extrémité, cette organisation est nommée *Little Endian*. À l'opposé, il existe des machines rangeant d'abord l'octet de poids fort, suivi de celui de poids faible. On les qualifie de *Big Endian*.

Figure 31.3
Stockage en mémoire de la valeur `0x1234`



Le programme suivant affiche la représentation en mémoire d'une valeur entière.

ordre_octets.c :

```
    #include <stdio.h>

    int
main (int argc, char * argv[])
{
```

```
    unsigned short int s_i;
    unsigned char *    ch;
    int                i;

    if ((argc != 2) || (sscanf(argv[1], "%hi", & s_i) != 1)) {
        fprintf(stderr, "Syntaxe : %s entier \n", argv [0]);
        exit(EXIT_FAILURE);
    }
    ch = (unsigned char *) & s_i;
    fprintf(stdout, "%04X représenté ainsi ", s_i);
    for (i = 0; i < sizeof(short int); i ++)
        fprintf(stdout, "%02X ", ch[i]);
    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution sur une machine *Little Endian*, un PC en l'occurrence :

```
$ ./ordre_octets 0x1234
1234 représenté ainsi 34 12
$
```

Et à présent sur un processeur Sparc *Big Endian*, on obtient :

```
$ ./ordre_octets 0x1234
1234 représenté ainsi 12 34
$
```

La différence peut même être encore plus accentuée avec des données sur 32 bits, car la valeur 0x12345678 peut être stockée sous 4 formes : 1234 5678, 5678 1234, 3412 7856, ou 7856 3412. Naturellement, la première et la dernière sont les plus fréquentes, mais rien n'interdit l'existence des autres.

Lorsque des données binaires doivent être écrites sur une machine et relues sur une autre, ce problème peut compliquer sérieusement le travail du développeur car cela interdit entre autres l'emploi de `fwrite()` et de `fread()`. Toutefois le problème reste au niveau de l'application, qui peut utiliser différentes techniques pour y remédier[1]. Là où la situation peut devenir vraiment gênante c'est lorsque des valeurs numériques sont employées dans les zones de données du protocole réseau lui-même. Par exemple la couche IP utilise une valeur numérique entière pour coder le protocole employé par la couche supérieure. Nous avons vu ces numéros dans la section précédente, dans le membre `p_proto` de la structure `protoent`. Lorsque la machine de destination reçoit un paquet de données au niveau de la couche IP, il faut qu'elle puisse décoder le numéro de protocole, par exemple 17, pour transmettre les données à la bonne couche de transport, UDP en l'occurrence. Le fait de devoir déterminer à chaque paquet l'ordre des octets de la machine émettrice est une surcharge de travail inacceptable.

La solution employée dans les protocoles fondés sur IP – et d'autres comme XNS – consiste à figer l'ordre des octets dans tous les en-têtes des paquets de données circulant sur le réseau. La forme retenue est *Big Endian*. Cela signifie que chaque machine doit convertir éventuellement l'ordre des données. L'avantage est que la bibliothèque C connaît à la compilation

1. Par exemple stocker une valeur connue comme `0x1234` en début de fichier et la lire pour déterminer l'ordre employé lors de l'enregistrement, c'est la méthode utilisée dans plusieurs formats graphiques.

l'ordre des octets sur la station où elle est installée, et qu'elle n'a pas de questions à se poser : soit il faut toujours convertir, soit il ne le faut jamais.

Pour le programmeur, cela implique toutefois une opération supplémentaire. Toutes les valeurs numériques qui seront transmises au protocole réseau devront passer par une étape de conversion éventuelle.

> **Attention**
> Nous parlons bien des valeurs transmises au *protocole* réseau. Les valeurs contenues dans les données de l'application ne sont pas concernées, quoique rien n'interdise d'employer le même mécanisme.

La bibliothèque C met à notre disposition quatre fonctions permettant de transformer un entier long ou court depuis l'ordre des octets de l'hôte vers celui du réseau, et inversement. Ces routines sont déclarées dans `<netinet/in.h>` :

```
unsigned long  int htonl (unsigned long  int valeur);
unsigned short int htons (unsigned short int valeur);
unsigned long  int ntohl (unsigned long  int valeur);
unsigned short int ntohs (unsigned short int valeur);
```

Les fonctions **`htonl()`** et **`htons()`** convertissent respectivement des entiers long et court depuis l'ordre des octets de l'hôte (`h`) vers (`to`) celui du réseau (*network* `n`). Parallèlement, **`ntohl()`** et **`ntohs()`** convertissent les entiers depuis l'ordre des octets du réseau vers celui de l'hôte. Nous emploierons les conversions d'entiers longs pour les adresses IP (qui sont sur 32 bits en version 4), et les conversions courtes pour les numéros de ports que nous allons voir dans la prochaine section.

Nous avons indiqué précédemment que le numéro de protocole indiqué dans le champ `p_proto` de la structure `protoent` était dans l'ordre des octets de l'hôte. Ceci nous a permis de les afficher directement avec `printf()` sans passer par une étape de conversion intermédiaire.

Enfin, remarquons qu'un programmeur qui doit choisir une plate-forme de développement aura intérêt à employer une architecture *Little Endian* (un PC sous Linux par exemple…) pour s'assurer de la portabilité de ses programmes. En effet, s'il oublie de convertir les données, sa machine n'étant pas du même type que le réseau, son programme échouera dès le début. Le bogue apparaîtra immédiatement lors de la mise au point, sans attendre un portage pour se révéler.

Services et numéros de ports

Lorsqu'un logiciel désire converser avec un correspondant qui est sur une autre machine, nous avons vu que le protocole IP, se trouvant sous les couches TCP ou UDP, permet de transmettre un paquet de données vers la station cible.

Toutefois, il peut y avoir beaucoup d'applications différentes qui fonctionnent simultanément sur la machine visée, et plusieurs d'entre elles peuvent offrir des fonctionnalités réseau. Il faut donc trouver le moyen de préciser quel correspondant nous désirons atteindre parmi les processus tournant sur l'ordinateur récepteur. Ceci est assuré par une fonctionnalité de la couche IP : les numéros de ports.

Chaque application voulant utiliser les services de la couche IP se verra affecter un numéro de port, c'est-à-dire un entier sur 16 bits qui permettra d'identifier le canal de communication au sein de la machine. C'est ici que se différencient les fonctionnalités de transmission de la

couche IP et celles de la couche UDP. Lorsqu'on envoie un paquet UDP, on lui affecte une adresse de destination mais également un numéro de port. La couche IP de la machine réceptrice passera les données à l'application associée à ce port.

Le processus émetteur est lui aussi doté d'un numéro de port, qui est d'ailleurs inscrit dans l'en-tête du paquet transmis, mais on s'intéresse généralement au numéro de port du récepteur plutôt qu'à celui de l'émetteur.

Les numéros de ports inférieurs à 255 sont strictement réservés à des services bien définis, disponibles sur de nombreux systèmes. Par exemple, le port 119 est réservé au service NNTP (*Network News Transfer Protocol*), c'est-à-dire au serveur Usenet. On peut se connecter directement avec l'utilitaire `telnet` en lui précisant le numéro de port :

```
$ telnet localhost 119
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
200 Leafnode NNTP Daemon, version 1.9.10 running at venux.ccb.fr
GROUP fr.comp.os.linux.annonces
211 43 2 44 fr.comp.os.linux.annonces group selected
HEAD 44
221 44 <ftp-20000308100005@Linux.EU.Org> article retrieved - head follows
Path: club-internet!grolier!freenix!enst!enst.fr!melchior.cuivre.fr.eu.org
!excalibur!fr.miroir!not-for-mail
Message-ID: <ftp-20000308100005@Linux.EU.Org>
From: ftpmaint@lip6.fr (Marc Victor)
Newsgroups: fr.comp.os.linux.annonces
Subject: [MIRROR] Nouveaux fichiers Linux sur ftp.lip6.fr
Date: 08 Mar 2000 10:00:05 +0100
References: <ftp-20000305100005@Linux.EU.Org>
Lines: 177
X-Posted-By: poste.sh version 1.1
Followup-To: poster
Approved: fcola@Linux.EU.Org
.
QUIT
205 Always happy to serve!
Connection closed by foreign host.
$
```

Les numéros de ports inférieurs à 1024 ne peuvent être associés qu'à des processus ayant un UID effectif nul ou la capacité **`CAP_NET_BIND_SERVICE`**. Ceci permet à un correspondant de savoir qu'il a bien affaire à un service officiel de la machine cible et pas à une application pouvant être faussée par un pirate. Les autres numéros peuvent être employés par n'importe quel utilisateur.

Précisons bien que les numéros de ports TCP et les numéros de ports UDP sont tout à fait distincts. On peut rencontrer simultanément des canaux de communication TCP et UDP ayant le même numéro tout en étant totalement indépendants.

On doit donc préciser pour joindre un processus distant :

- l'adresse IP de la machine sur laquelle il s'exécute ;
- le numéro de port de réception ;
- le protocole (UDP ou TCP) employé.

Lorsqu'une machine dispose de plusieurs interfaces réseau (plusieurs cartes, ou une carte et une liaison PPP par exemple), les numéros de ports sur chaque interface sont également indépendants.

Pour connaître l'association entre un numéro de port et un service particulier, on peut consulter le fichier `/etc/services`. Celui-ci contient les numéros attribués à environ 300 services courants. Ce fichier est consultable par tous les utilisateurs :

```
$ cat /etc/services
# /etc/services:
# $Id: services,v 1.4 1997/05/20 19:41:21 tobias Exp $
#
# Network services, Internet style
#
tcpmux          1/tcp                       # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp       sink null
discard         9/udp       sink null
systat          11/tcp      users
    [...]
ftp             21/tcp
fsp             21/udp      fspd
ssh             22/tcp                      # SSH Remote Login Protocol
ssh             22/udp                      # SSH Remote Login Protocol
telnet          23/tcp
# 24 - private
smtp            25/tcp      mail
# 26 - unassigned
time            37/tcp      timserver
time            37/udp      timserver
    [...]
fido            60179/tcp                   # Ifmail
fido            60179/udp                   # Ifmail

# Local services

linuxconf       98/tcp
$
```

Sur certains systèmes, le fichier `/etc/services` est complété par des données provenant d'un serveur NIS, accessible avec `ypcat -k services`.

On remarque que certains services offrent à la fois une interface en TCP et en UDP, alors que d'autres ne travaillent que dans un seul mode. Il peut exister des alias, par exemple le service de messagerie `smtp` peut être aussi invoqué sous le nom `mail`.

```
$ telnet localhost smtp
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 venux.ccb.fr ESMTP Sendmail 8.9.3/8.9.3;
QUIT
221 venux.ccb.fr closing connection
```

```
Connection closed by foreign host.
$ telnet localhost mail
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 venux.ccb.fr ESMTP Sendmail 8.9.3/8.9.3;
QUIT
221 venux.ccb.fr closing connection
Connection closed by foreign host.
$
```

Pour créer une socket de communication, il est nécessaire de connaître le numéro de port à utiliser. Toutefois, on ne peut pas demander à l'utilisateur d'indiquer lui-même la valeur numérique. Il faut l'autoriser à employer un mot-clé indiqué dans le fichier `/etc/services`, même s'il s'agit d'une application locale. Comme pour le fichier des protocoles, il existe des routines de bibliothèque pour nous aider à rechercher des services sur le système.

La fonction **`getservbyname()`**, déclarée dans `<netdb.h>`, permet de retrouver un service à partir de son nom ou d'un alias :

```
struct servent * getservbyname (const char * nom, const char * protocole);
```

Cette routine prend en premier argument une chaîne de caractères indiquant le nom du service, par exemple « `ftp` », et en seconde position une chaîne de caractères mentionnant le protocole concerné, comme nous l'avons déterminé dans la section précédente.

La fonction **`getservbyport()`** permet de chercher un service à partir du numéro de port indiqué dans l'ordre des octets du réseau.

```
struct servent * getservbyport (short int numero, const char * protocole);
```

La structure **`servent`** fournie par ces routines est définie ainsi :

Nom	Type	Signification
`s_name`	`char *`	Nom officiel du service défini dans la RFC 1700
`s_port`	`short int`	Numéro du service dans l'ordre des octets du réseau
`s_proto`	`char *`	Nom du protocole associé
`s_aliases`	`char **`	Liste éventuelle d'alias, terminée par un pointeur `NULL`

Le programme suivant affiche les résultats concernant les services indiqués sur sa ligne de commande. Ceux-ci peuvent être fournis sous forme numérique ou par un nom.

exemple_getservby.c :

```
    #include <stdio.h>
    #include <netinet/in.h>
    #include <netdb.h>

    void affiche_service (char * nom, char * proto);

    int
main (int argc, char * argv[])
```

```
{
    int i;
    for (i = 1; i < argc; i ++) {
        affiche_service(argv[i], "tcp");
        affiche_service(argv[i], "udp");
    }
    return EXIT_SUCCESS;
}

    void
affiche_service (char * nom, char * proto)
{
    int              i;
    int              port;
    struct servent * service;
    if (sscanf(nom, "%d", & port) == 1)
        service = getservbyport(htons(port), proto);
    else
        service = getservbyname(nom, proto);
    if (service == NULL) {
        fprintf(stdout, "%s / %s : inconnu \n", nom, proto);
    } else {
        fprintf(stdout, "%s / %s : %s ( ",
                nom, proto, service->s_name);
        for (i = 0; service->s_aliases[i] != NULL; i ++)
            fprintf(stdout, "%s ", service->s_aliases[i]);
        fprintf(stdout, ") port = %d\n", ntohs(service->s_port));
    }
}
```

Comme nous ne précisons pas sur la ligne de commande le protocole employé, le programme essaye successivement TCP et UDP.

```
$ ./exemple_getservby mail 21
mail / tcp : smtp ( mail ) port = 25
mail / udp : inconnu
21 / tcp : ftp ( ) port = 21
21 / udp : fsp ( fspd ) port = 21
$ ./exemple_getservby NNTP nntp
NNTP / tcp : inconnu
NNTP / udp : inconnu
nntp / tcp : nntp ( readnews untp ) port = 119
nntp / udp : inconnu
$
```

Il est possible aussi de parcourir entièrement la liste des services connus par le système à l'aide des routines **setservent()**, **getservent()** et **endservent()**, qui ont un comportement similaire à `setprotoent()`, `getprotoent()` et `endprotoent()`.

```
void             setservent (int ouvert);
struct servent * getservent (void);
void             endservent (void);
```

Attention, le fait qu'un service soit connu par le système ne signifie pas qu'il y ait un processus prêt à lui répondre. Par exemple le service `systat` fournit des informations qui peuvent renseigner un éventuel pirate, aussi le désactive-t-on souvent :

```
$ ./exemple_getservby systat
systat / tcp : systat ( users ) port = 11
systat / udp : inconnu
$ telnet localhost systat
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connexion refusée
$
```

Mentionnons l'existence de fonctions réentrantes, sous forme d'extensions Gnu :

```
int getservbyport_r    (short int numero,
                        struct servent * service,
                        char * buffer, size_t taille_buffer,
                        struct servent ** retour);
int getservbyname_r    (const char * nom,
                        struct servent * service,
                        char * buffer, size_t taille_buffer,
                        struct servent ** retour);
int getservent_r       (struct servent * service,
                        char * buffer, size_t taille_buffer,
                        struct servent ** retour);
```

Voici un exemple de balayage de tous les services.

exemple_getservent_r.c :

```
    #define _GNU_SOURCE
    #include <stdio.h>
    #include <netdb.h>

    int
main (void)
{
    struct servent   service;
    struct servent * retour;
    char             buffer[256];

    setservent(0);
    while (getservent_r(& service, buffer, 256, & retour) == 0)
        fprintf(stdout, "%s ", service.s_name);
    endservent();
    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

Manipulation des adresses IP

Le protocole IP version 4 offre le routage des paquets à destination d'hôtes dont le nom est indiqué par une adresse sur 32 bits. Nous avons déjà précisé que cette adresse est souvent écrite dans la notation pointée, en séparant les octets et en les affichant en décimal. La valeur

est donc contenue dans un `unsigned long int`. Néanmoins les fonctions de communication disponibles, que nous verrons dans le prochain chapitre, sont utilisables sur d'autres supports que le protocole IPv4, ne serait-ce que l'IPv6 qui est déjà disponible à titre expérimental et réclame des adresses sur 48 bits. On peut aussi vouloir faire communiquer des processus résidant tous sur la même machine, et les mêmes routines de communication sont disponibles avec un adressage réalisé par l'intermédiaire de noms de fichiers.

La définition d'une adresse est donc assurée par une structure qui peut prendre des formes diverses suivant le protocole employé. Pour les communications IPv4, on emploie la structure **`in_addr`**, définie dans `<netinet/in.h>` avec un seul membre :

Nom	Type	Signification
`s_addr`	`unsigned long int`	Adresse IP dans l'ordre des octets du réseau

La structure **`in6_addr`** utilisée pour les adresses IPv6 est plus compliquée puisqu'elle se présente sous forme d'`union`. Nous la considérerons comme un type opaque.

Il existe des fonctions, déclarées dans `<arpa/inet.h>`, permettant de convertir directement l'adresse `in_addr` en notation pointée, et inversement. La routine **`inet_ntoa()`** (*Network to Ascii*) renvoie une chaîne de caractères statiques représentant en notation pointée l'adresse transmise en argument.

```
char * inet_ntoa (struct in_addr adresse);
```

Attention
On passe bien la valeur de la structure, et pas un pointeur.

Il n'y a pas d'équivalent réentrant. La documentation Gnu précise que chaque thread dispose de son propre buffer et qu'il n'y a donc pas de problèmes d'accès concurrents. Toutefois pour assurer la portabilité sur d'autres plates-formes, on emploiera dans un programme multithread un sémaphore pour organiser les appels simultanés et l'accès à la mémoire statique[1].

La fonction inverse, **`inet_aton()`**, remplit la structure `in_addr` passée en second argument en ayant converti la chaîne pointée transmise en première position. Si cette chaîne est invalide, `inet_aton()` renvoie 0.

```
int inet_aton (const char * chaine, struct in_addr * adresse);
```

Cette fonction n'est malheureusement disponible que sur peu de systèmes. Lors d'un portage, on peut employer **`inet_addr()`**, qui prend en argument la chaîne en notation pointée et renvoie directement l'adresse sous forme d'entier long non signé.

```
unsigned long int inet_addr (const char * chaine);
```

Le problème est qu'en cas d'échec cette routine renvoie une valeur particulière `INADDR_NONE`, qui peut aussi représenter une adresse valide : `255.255.255.255`. On l'utilisera donc uniquement si `inet_aton()` n'est pas disponible.

1. On notera aussi qu'on ne peut pas afficher les résultats de plusieurs `inet_ntoa()` successifs dans le même `printf()`.

Voici un programme qui emploie les arguments en ligne de commande avec ces trois routines :

exemple_inet_aton.c :

```
    #include <stdio.h>
    #include <arpa/inet.h>
    #include <netinet/in.h>

    int
main (int argc, char * argv[])
{
    struct in_addr adresse;
    int            i;

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "inet_aton (%s) = ", argv [i]);
        if (inet_aton(argv[i], & adresse) == 0) {
            fprintf(stdout, "invalide \n");
            continue;
        }
        fprintf(stdout, "%08X \n", ntohl(adresse . s_addr));
        fprintf(stdout, "inet_addr(%s) = ", argv[i]);
        if ((adresse.s_addr = inet_addr(argv[i])) == INADDR_NONE) {
            fprintf(stdout, "invalide \n");
            continue;
        }
        fprintf(stdout, "%08X \n", ntohl(adresse.s_addr));
        fprintf(stdout, "inet_ntoa(%08X) = %s \n",
                            ntohl(adresse.s_addr),
                            inet_ntoa(adresse));
    }
    return EXIT_SUCCESS;
}
```

Nous remarquons bien le problème que pose `inet_addr()` par rapport à `inet_aton()` :

```
$ ./exemple_inet_aton 172.16.15.1
inet_aton(172.16.15.1) = AC100F01
inet_addr(172.16.15.1) = AC100F01
inet_ntoa(AC100F01) = 172.16.15.1
$ ./exemple_inet_aton 255.255.255.255
inet_aton(255.255.255.255) = FFFFFFFF
inet_addr(255.255.255.255) = invalide
$
```

La constante `INADDR_NONE` a la même valeur que `INADDR_BROADCAST`, qui correspond à une adresse (`255.255.255.255`) employée pour diffuser un message vers tous les hôtes accessibles. Nous reviendrons sur ce sujet ultérieurement.

Les adresses IP sont organisées en sous-réseaux, afin de simplifier les routages. Chaque sous-réseau possède une adresse, et chaque machine a elle-même une adresse au sein du sous-réseau. L'adresse IP complète est obtenue en faisant suivre l'adresse du sous-réseau par celle de la station. Ainsi, si on a un sous-réseau `192.1.1`, sa station numéro `2` aura l'adresse `192.1.1.2`. Cette organisation permet à un routeur de savoir sur quel brin Ethernet se trouve un hôte sous sa responsabilité.

Les sous-réseaux définis par le NIC (*Network Information Center*), qui gère les adressages sur Internet, sont répartis en plusieurs catégories :

- Les 127 sous-réseaux de classe A ont des adresses allant de `1.` à `127.` Chacun d'eux peut adresser ses stations sur 24 bits, ce qui leur permet de regrouper plus d'un million de machines.
- Les adresses des sous-réseaux de classe B vont de `128.0.` à `191.255`. Il y en a plus de seize mille, chacun pouvant contenir des machines avec des adresses sur 16 bits. Il y a un peu plus de 65 000 stations adressables par réseau.
- Les sous-réseaux de classe C s'étendent de `192.0.0.` à `223.255.255`. Chacun de ces deux millions de sous-réseaux peut contenir 254 hôtes, car les adresses `.0` et `.255` ne sont pas utilisables.
- La classe D s'étend de `224.0.0.0` à `239.255.255.255`. Il ne s'agit pas d'adresses de machines, mais uniquement d'adresses de diffusion *multicast*. Nous détaillerons ce concept dans le prochain chapitre.

Il existe d'autres classes pour les adresses supérieures à `240.`, mais il s'agit uniquement d'utilisations expérimentales.

Les deux fonctions **`inet_netof()`** et **`inet_lnaof()`** `sont` capables d'extraire respectivement la partie réseau et la partie adresse locale d'une adresse IP complète.

```
unsigned long int inet_netof (struct in_addr);
unsigned long int inet_lnaof (struct in_addr);
```

Elles renvoient un entier long non signé dans l'ordre des octets de l'hôte.

exemple_inet_netof.c :

```
    #include <stdio.h>
    #include <arpa/inet.h>
    #include <netinet/in.h>

    int
main (int argc, char * argv[])
{
    int i;
    struct in_addr adresse;
    unsigned long int reseau;
    unsigned long int locale;

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "inet_netof(%s) = ", argv[i]);
        if (inet_aton(argv[i], & adresse) == 0) {
            fprintf(stdout, "invalide \n");
            continue;
        }
        reseau = inet_netof(adresse);
        locale = inet_lnaof(adresse);
        fprintf(stdout, "%08lX + %08lX\n", reseau, locale);
    }
    return EXIT_SUCCESS;
}
```

Nous allons l'essayer successivement avec des adresses de classe A, B et C :

```
$ ./exemple_inet_netof 1.2.3.4
inet_netof (1.2.3.4) = 00000001 + 00020304
$ ./exemple_inet_netof 128.2.3.4
inet_netof (128.2.3.4) = 00008002 + 00000304
$ ./exemple_inet_netof 192.2.3.4
inet_netof (192.2.3.4) = 00C00203 + 00000004
$
```

On peut remarquer que `inet_netof()` extrait vraiment l'adresse du sous-réseau et n'effectue pas simplement un masque. On peut toutefois avoir besoin de cette fonctionnalité pour présenter les résultats à l'utilisateur. On peut l'implémenter ainsi :

masque_reseau.c :

```
    #include <stdio.h>
    #include <arpa/inet.h>
    #include <netinet/in.h>

    int
main (int argc, char * argv[])
{
    int i;
    struct in_addr adresse;
    struct in_addr reseau;
    struct in_addr locale;

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "%s : \n", argv[i]);
        if (inet_aton(argv[i], & adresse) == 0) {
            fprintf(stdout, "  invalide \n");
            continue;
        }
        adresse.s_addr = ntohl(adresse . s_addr);
        if (adresse.s_addr < 0x80000000L) {
            reseau.s_addr = adresse.s_addr & 0xFF000000L;
            locale.s_addr = adresse.s_addr & 0x00FFFFFFL;
        } else if (adresse.s_addr < 0xC0000000L) {
            reseau.s_addr = adresse.s_addr & 0xFFFF0000L;
            locale.s_addr = adresse.s_addr & 0x0000FFFFL;
        } else {
            reseau.s_addr = adresse.s_addr & 0xFFFFFF00L;
            locale.s_addr = adresse.s_addr & 0x000000FFL;
        }
        reseau.s_addr = htonl(reseau.s_addr);
        locale.s_addr = htonl(locale.s_addr);
        fprintf(stdout, "  adresse reseau = %s\n", inet_ntoa(reseau));
        fprintf(stdout, "  adresse locale = %s\n", inet_ntoa(locale));
    }
    return EXIT_SUCCESS;
}
```

Cette fois-ci les affichages correspondent bien aux diverses parties de l'adresse IP complète :

```
$ ./masque_reseau 1.2.3.4
1.2.3.4 :
  adresse reseau = 1.0.0.0
  adresse locale = 0.2.3.4
$ ./masque_reseau 172.16.15.1
172.16.15.1 :
  adresse reseau = 172.16.0.0
  adresse locale = 0.0.15.1
$ ./masque_reseau 192.1.1.20
192.1.1.20 :
  adresse reseau = 192.1.1.0
  adresse locale = 0.0.0.20
$
```

Les fonctions `inet_ntoa()` et `inet_aton()` présentent le défaut d'être liées aux adresses IP version 4. Dans le protocole IPv6, les adresses sont représentées sur 128 bits, qu'on écrit sous forme de huit valeurs hexadécimales sur 16 bits, séparées par des deux-points (par exemple `4235:1a05:0653:5d48:1b94:5710:32c4:ae25`).

Les deux fonctions **`inet_ntop()`** et **`inet_pton()`** offrent les mêmes services, mais en étant prêtes pour l'adressage IPv6. Le `p` signifie *présentation*.

```
const char * inet_ntop (int famille, const void * adresse,
                        char * buffer, size_t longueur);
```

Le premier argument doit être `AF_INET` si on utilise une adresse IPv4 ou `AF_INET6` pour une IPv6. Le second argument pointe vers une structure `in_addr` en IPv4, ou `in6_addr` en IPv6.

```
int inet_pton (int famille, const char * chaine, void * adresse)
```

Comme pour `inet_ntop()`, on indique la famille (`AF_INET` ou `AF_INET6`) en premier argument, et on donne un pointeur vers la structure `in_addr` ou `in6_addr` en dernière position. Le programme suivant essaye successivement les deux types de conversion sur ses arguments en ligne de commande.

exemple_inet_pton.c :

```
    #include <stdio.h>
    #include <arpa/inet.h>
    #include <netinet/in.h>

    int
main (int argc, char * argv[])
{
    struct in6_addr adresse_6;
    struct in_addr  adresse_4;
    int             i;
    char            buffer [256];
    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "inet_pton(%s) = ", argv[i]);
        if (inet_pton(AF_INET6, argv[i], & adresse_6) != 0) {
            fprintf(stdout, "IPv6 : ");
```

```
            inet_ntop(AF_INET6, & adresse_6, buffer, 256);
            fprintf(stdout, "inet_ntop() = %s \n", buffer);
            continue;
        }
        if (inet_pton(AF_INET, argv[i], & adresse_4) != 0) {
            fprintf(stdout, "IPv4 : ");
            inet_ntop(AF_INET, & adresse_4, buffer, 256);
            fprintf(stdout, "inet_ntop() = %s \n", buffer);
            continue;
        }
        fprintf(stdout, "invalide \n");
    }
    return EXIT_SUCCESS;
}
```

On peut convertir correctement les deux types d'adresses.

```
$ ./exemple_inet_pton 192.1.1.10
inet_pton(192.1.1.10) = IPv4 : inet_ntop() = 192.1.1.10
$ ./exemple_inet_pton ::2
inet_pton(::2) = IPv6 : inet_ntop() = ::2
$ ./exemple_inet_pton 4235:1a05:0653:5d48:1b94:5710:32c4:ae25
inet_pton (4235:1a05:0653:5d48:1b94:5710:32c4:ae25) = IPv6 : inet_ntop ()
➥ = 4235:1a05:653:5d48:1b94:5710:32c4:ae25
$
```

Attention

Ces fonctions ne sont pas très répandues, leur portabilité est loin d'être assurée.

Noms d'hôtes et noms de réseaux

Les adresses IP des machines ne sont pas faciles à mémoriser (surtout en IPv6 !). Pour simplifier la vie des administrateurs et des utilisateurs, on associe donc des noms aux stations. Lorsqu'il n'y a que deux ou trois machines sur un même réseau, on inscrit simplement les correspondances dans le fichier `/etc/hosts`. Cependant, dès qu'on dépasse une dizaine de stations, la maintenance de tous les fichiers `hosts` se complique car il faut tous les modifier dès qu'une machine est ajoutée.

On emploie alors un serveur de noms, c'est-à-dire un logiciel capable de répondre à des requêtes pour obtenir l'adresse d'une machine dont on connaît le nom. La base de données est alors centralisée en un seul point sous le contrôle de l'administrateur. Le serveur de noms est aussi capable – lorsqu'il ne peut pas répondre – d'indiquer l'adresse d'un autre serveur mieux qualifié pour traiter la demande.

L'interrogation du serveur de noms se fait à l'aide de routines de bas niveau assez complexes, dont nous ne parlerons pas ici. Heureusement, la bibliothèque C offre des routines permettant de rechercher facilement un hôte à partir de son nom ou de son adresse.

Les informations concernant un hôte sont regroupées dans une structure **hostent** définie ainsi :

Nom	Type	Signification
h_name	char *	Nom d'hôte officiel.
h_aliases	char * *	Liste d'alias, terminée par un pointeur NULL.
h_addrtype	int	Type d'adresse, AF_INET ou AF_INET6.
h_length	int	Longueur des adresses du type indiqué ci-dessus, en octets.
h_addr_list	char **	Liste d'adresses correspondant à cet hôte (il peut y avoir plusieurs interfaces réseau et le même nom sur chacune d'elles). Les adresses sont données dans l'ordre des octets du réseau.
h_addr	char *	Équivalent de h_addr_list [0].

Les adresses de la liste h_addr_list[] ne sont pas des chaînes de caractères mais des blocs de mémoire qu'on pourra convertir en structure in_addr ou in6_addr.

Les fonctions **gethostbyname()** et **gethostbyaddr()** permettent de retrouver les informations concernant un hôte à partir de son nom ou de son adresse IP. Il existe des fonctions équivalentes réentrantes, sous forme d'extensions Gnu, **gethostbyname_r()** et **gethostbyaddr_r()**.

```
struct hostent * gethostbyname (const char * nom);
struct hostent * gethostbyaddr (const char * adresse,
                                int longueur, int format);
int gethostbyname_r (const char * nom,
                     struct hostent * hote,
                     char * buffer, size_t taille_buffer
                     int * erreur);
int gethostbyaddr_r (const char * adresse,
                     int longueur, int format);
                     struct hostent * hote,
                     char * buffer, size_t taille_buffer
                     int * erreur);
```

L'argument format de gethostbyaddr() correspond à AF_INET ou AF_INET6. Le dernier argument des fonctions réentrantes est un pointeur dans lequel on mémorisera les conditions d'erreur. On détaillera ceci plus bas.

L'extension Gnu **gethostbyname2()** et son équivalent **gethostbyname2_r()** permettent de restreindre le champ de recherche en indiquant la famille IP désirée :

```
struct hostent * gethostbyname2 (const char * nom, int famille);
int gethostbyname2_r (const char * nom, int famille,
                      struct hostent * hote,
                      char * buffer, size_t taille_buffer
                      int * erreur);
```

Le programme suivant va permettre d'obtenir des informations sur les noms ou adresses d'hôtes passés en ligne de commande. Il essaye successivement de les lire comme une adresse IPv4, une adresse IPv6, et un nom d'hôte. Dans tous les cas, il recherche la structure hostent correspondante et affiche les résultats.

exemple_gethostby.c :

```
    #include <stdio.h>
    #include <arpa/inet.h>
    #include <netdb.h>
    #include <netinet/in.h>
    #include <sys/types.h>

    int
main (int argc, char * argv [])
{
    int               i, j;
    struct hostent *  hote;
    struct in_addr    adresse_4;
    struct in6_addr   adresse_6;
    struct in_addr *  ip_4;
    struct in6_addr * ip_6;
    char              buffer [256];

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "%s : ", argv[i]);
        /* Vérifions d'abord s'il s'agit d'une adresse pointée IPv4 */
        if (inet_aton(argv[i], & adresse_4) != 0) {
            /* On récupère la structure hostent */
            if ((hote = gethostbyaddr((char *) & adresse_4,
                          sizeof(struct in_addr), AF_INET)) == 0) {
                fprintf(stdout, "??? \n");
                continue;
            }
        /* Sinon on recherche une adresse IPv6 */
        } else if (inet_pton(AF_INET6, argv[i], & adresse_6) != 0) {
            if ((hote = gethostbyaddr((char *) & adresse_6,
                          sizeof(struct in6_addr), AF_INET6)) == 0) {
                fprintf(stdout, "??? \n");
                continue;
            }
        } else {
            /* On interroge la résolution de noms */
            if ((hote = gethostbyname(argv[i])) == NULL) {
                fprintf(stdout, "??? \n");
                continue;
            }
        }
        /* On peut afficher le contenu de la structure hostent */
        fprintf(stdout, "%s (", hote->h_name);
        for (j = 0; hote->h_aliases[j] != NULL; j ++)
            fprintf(stdout, " %s", hote->h_aliases[j]);
        fprintf(stdout, " ) : ");
        if (hote->h_addrtype == AF_INET6) {
            for (j = 0; hote->h_addr_list[j] != NULL; j ++) {
                ip_6 = (struct in6_addr *) (hote->h_addr_list[j]);
                inet_ntop(AF_INET6, ip_6, buffer, 256);
                fprintf(stdout, "%s ", buffer);
```

```
            }
        } else {
            for (j = 0; hote->h_addr_list[j] != NULL; j ++) {
                ip_4 = (struct in_addr *) (hote->h_addr_list[j]);
                fprintf(stdout, "%s ", inet_ntoa(* ip_4));
            }
        }
        fprintf(stdout, "\n");
    }
    return EXIT_SUCCESS;
}
```

Naturellement, la résolution s'effectue en utilisant le fichier `/etc/hosts` local mais aussi en interrogeant les serveurs de noms.

```
$ ./exemple_gethostby localhost
localhost : localhost ( localhost.localdomain ) : 127.0.0.1
$ ./exemple_gethostby venux
venux : venux ( venux.ccb.fr ) : 192.1.1.51
$ ./exemple_gethostby 192.1.1.51
192.1.1.51 : venux ( venux.ccb.fr ) : 192.1.1.51
$ ./exemple_gethostby ftp.lip6.fr
ftp.lip6.fr : nephtys.lip6.fr ( ftp.lip6.fr ) : 195.83.118.1
$ ./exemple_gethostby sunsite.unc.edu
sunsite.unc.edu : sunsite.unc.edu ( ) : 152.2.254.81
$ ./exemple_gethostby 195.36.208.117
195.36.208.117 : Corbeil-2-117.club-internet.fr ( ) : 195.36.208.117
$
```

Nous avons ajouté pas mal de code dans le programme, uniquement pour être capable de traiter les adresses IPv6 alors qu'elles ne sont pas encore employées. On peut simplifier largement le travail pour une application ne désirant pas les gérer.

La bibliothèque C nous offre également des routines pour consulter l'ensemble de la base de données des hôtes se trouvant dans le même domaine que le nôtre. Les fonctions **`sethostent()`**, **`gethostent()`** et **`endhostent()`** ont un comportement similaire à `setservent()`, `getservent()` et `endservent()` que nous avons déjà observées :

```
void             sethostent (int ouvert);
struct hostent * gethostent (void);
void             endhostent (void);
```

exemple_gethostent.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <netdb.h>

    int
main (void)
{
    struct hostent * hote;
    sethostent(1);
    while ((hote = gethostent()) != NULL)
        fprintf(stdout, "%s ", hote->h_name);
```

```
    printf(stdout, "\n");
    endhostent();
    return EXIT_SUCCESS;
}
```

L'exécution affiche la liste des machines du même domaine :

```
$ ./exemple_gethostent
localhost venux tracy gimli visux jcv matisse ifr-cdg ifr-orly
    [...]
$
```

Indiquons pour finir qu'il existe une base de données des sous-réseaux, moins connue que la base des noms d'hôtes, car elle est plutôt employée pour l'administration du système qu'au quotidien. Cette base de données est souvent constituée par le fichier `/etc/network`. Les fonctions d'accès manipulent des structures `netent`, définies ainsi :

Nom	Type	Signification
`n_name`	`char *`	Nom du sous-réseau
`n_aliases`	`char **`	Liste d'alias terminée par un pointeur `NULL`
`n_addrtype`	`Int`	Type d'adresses sur le sous-réseau (uniquement `AF_INET` pour le moment)
`n_net`	`unsigned long int`	Adresse du sous-réseau, dans l'ordre des octets de l'hôte

Les fonctions d'accès sont **`getnetbyname()`**, **`getnetbyaddr()`** et, pour balayer la base des sous-réseaux, on emploie **`setnetent()`**, **`getnetent()`** et **`endnetent()`**.

```
struct netent * getnetbyname (const char * nom);
struct netent * getnetbyaddr (unsigned long int adresse, int type);
void            setnetent (int ouvert);
struct netent * getnetent (void);
void            endnetent (void);
```

Gestion des erreurs

Les fonctions d'accès à la base de données des hôtes n'emploient pas directement `errno` mais une autre variable globale, **`h_errno`**, déclarée dans `<netdb.h>`. Dans la bibliothèque GlibC, cette variable est dupliquée pour chaque thread et peut donc être employée sans danger dans un programme multithread.

Les codes d'erreur qu'on peut y trouver au retour de toutes les routines `gethostXXX()` sont les suivants :

Nom	Signification
`NETDB_SUCCES`	Pas d'erreur.
`HOST_NOT_FOUND`	L'hôte n'a pas été trouvé.
`TRY_AGAIN`	Un problème temporaire de serveur de noms est apparu. On peut réitérer la demande.
`NO_RECOVERY`	Une erreur critique est apparue durant la résolution du nom.
`NO_ADRESS`	L'hôte est connu du serveur de noms, mais il n'a pas d'adresse valide.

Les fonctions **herror()** et **hstrerror()** sont considérées comme obsolètes. Elles sont l'équivalent de perror() et strerror(), appliquées à h_errno.

```
void         herror    (const char * chaine);
const char * hstrerror (int erreur);
```

Conclusion

Nous avons examiné dans ce chapitre l'essentiel des moyens d'accès aux bases de données permettant la résolution de noms – tant au niveau des hôtes que des services et des protocoles. Une grosse partie du travail dans les programmes réseau repose sur ces routines.

Pour obtenir plus d'informations, on peut consulter bien entendu [STEVENS 1990] *UNIX Network Programming* – qui est un grand classique de ce domaine.

En ce qui concerne l'installation et l'administration d'un réseau sous Linux, on conseillera à nouveau [KIRCH 2001] *L'administration réseau sous Linux*, et le *NET-3-HOWTO*.

Enfin, on trouvera ci-dessous quelques références des documents RFC définissant les principaux protocoles utilisés sur Internet.

Numéro RFC	Auteur et date	Sujet
RFC 791	J. Postel, 01/09/1981	IP : Internet Protocol.
RFC 792	J. Postel, 01/09/1981	ICMP : Internet Control Message Protocol.
RFC 793	J. Postel, 01/09/1981	TCP : Transmission Control Protocol.
RFC 768	J. Postel, 28/08/1980	UDP : User Datagram Protocol.
RFC 959	J. Postel, J. Reynolds, 01/10/1985	FTP : File Transfert Protocol.
RFC 783	K.R. Sollins, 01/06/1981	TFTP : Trivial File Transfert Protocol.
RFC 821	J. Postel, 01/08/1982	SMTP : Simple Mail Transfert Protocol.
RFC 977	B. Kantor, 01/02/1986	NNTP : Network News Transfer Protocol.
RFC 854	J. Postel, J. Reynolds, 01/05/1983	Telnet Protocol.
RFC 1918	Y. Rekhter et al. 01/02/1996	Adresses IP utilisables sur un réseau privé.
RFC 2500	J. Reynolds, 01/06/1999	Protocoles standard sur Internet.
RFC 1700	J. Reynolds, J. Postel, 01/10/1994	Noms et numéros standard pour tout ce qui concerne les communications sur Internet.

À présent que nous savons déterminer l'adresse d'un correspondant et le service qui nous intéressent, nous allons pouvoir enfin établir la communication avec un processus distant dans le prochain chapitre.

32

Utilisation des sockets

Concept de socket

Les sockets[1] sont apparues dans 4.2BSD, en 1983. Elles sont à présent disponibles sur tous les Unix courants, et il en existe des variantes sur les autres principaux systèmes d'exploitation. Il s'agit approximativement d'une extension de la portée des tubes nommés, pour pouvoir faire dialoguer des processus s'exécutant sur différentes machines.

On peut donc écrire des données dans une socket après l'avoir associée à un protocole de communication, et les couches réseau des deux stations s'arrangeront pour que les données ressortent à l'autre extrémité. La seule complication introduite par rapport aux tubes classiques est la phase d'initialisation, car il faut indiquer l'adresse et le numéro de port du correspondant. Une fois que la liaison est établie, le comportement ne sera pas très différent de ce qu'on a étudié dans le chapitre 28.

Les sockets sont représentées dans un programme par des entiers, comme les descripteurs de fichiers. On peut leur appliquer les appels-système usuels, `read()`, `write()`, `select()`, `close()`, etc. Nous verrons dans ce chapitre les primitives spécifiques qui s'appliquent aux sockets.

Création d'une socket

La première étape consiste à créer une socket. Ceci s'effectue à l'aide de l'appel-système **`socket()`**, défini dans `<sys/socket.h>` :

```
int socket (int domaine, int type, int protocole);
```

1. Le mot socket se traduit par *prise* en français (dans le sens de prise de courant), mais j'utiliserai le mot original, qui est devenu un terme consacré.

Le premier argument de cette routine est le domaine de communication. Il s'agit d'une constante symbolique pouvant prendre plusieurs valeurs. En voici quelques exemples :

- `AF_INET` : protocole fondé sur IP.
- `AF_INET6` : protocole IPng, expérimental et soumis à des options de compilation dans le noyau.
- `AF_UNIX` : communication limitée aux processus résidant sur la même machine. Dans certains cas cette constante est remplacée par le synonyme `AF_LOCAL`, qui appartient d'ailleurs à la terminologie Posix.

Nous ne nous intéresserons ici qu'au domaine `AF_INET`, qui regroupe toutes les communications réseau avec IP, TCP, UDP ou ICMP.

Le second argument est le type de socket. Nous ne considérerons que trois cas :

- `SOCK_STREAM` : le dialogue s'effectue en mode connecté, avec un contrôle de flux d'une extrémité à l'autre de la communication.
- `SOCK_DGRAM` : la communication a lieu sans connexion, par transmission de paquets de données.
- `SOCK_RAW` : la socket sera utilisée pour dialoguer de manière brute avec le protocole.

Nous reviendrons ultérieurement sur les communications en mode connecté ou non. Finalement, le troisième argument indique le protocole désiré. Il s'agit du champ `p_proto` de la structure `protoent` examinée dans le chapitre précédent. Si on indique une valeur nulle, les combinaisons suivantes seront automatiquement réalisées :

Domaine	Type	Socket obtenue	Protocole équivalent
`AF_INET`	`SOCK_STREAM`	Socket de dialogue avec le protocole TCP/IP	`IPPROTO_TCP`
`AF_INET`	`SOCK_DGRAM`	Socket utilisant le protocole UDP/IP	`IPPROTO_UDP`

Pour les sockets de type `SOCK_RAW`, il faut utiliser l'un des deux protocoles suivants :

- `IPPROTO_RAW` : communication directe avec la couche IP.
- `IPPROTO_ICMP` : communication utilisant le protocole ICMP. Ceci est utilisé par exemple dans l'utilitaire `/bin/ping`.

La création d'une socket de type `SOCK_RAW` nécessite la capacité `CAP_NET_RAW`. Les utilitaires qui en emploient (`traceroute`, `ping`…) et qui sont ouverts à tous les utilisateurs (à la différence de `tcpdump`) sont donc normalement installés Set-UID *root*.

La création d'une socket à l'aide de l'appel-système éponyme ne fait que réserver un emplacement dans la table des descripteurs du noyau. Au retour de cette routine, nous disposons d'un entier permettant de distinguer la socket, mais aucun dialogue réseau n'a pris place. Il n'y a même pas eu d'échange d'informations avec les protocoles de communication du noyau. Celui-ci a simplement accepté de nous attribuer un emplacement dans sa table de sockets.

Le descripteur est supérieur ou égal à zéro, ce qui signifie qu'une valeur de retour négative indique une erreur. Comme aucune communication n'a commencé, les seules erreurs possibles sont :

- `EINVAL`, si le domaine est invalide.

- EPROTONOSUPPORT, si le type est incohérent avec le protocole ou le domaine.
- EACCES, si on n'a pas l'autorisation de créer une socket du type demandé (par exemple AF_INET et SOCK_RAW).

À cela s'ajoutent comme toujours EMFILE, ENFILE, ENOMEM si l'espace disponible dans la mémoire du noyau est insuffisant.

Avant de pouvoir l'utiliser, il faut *identifier* la socket, c'est-à-dire définir l'adresse complète de notre extrémité de communication. Le nom affecté à la socket doit permettre de la trouver sans ambiguïté en employant le protocole réseau indiqué lors de sa création. Pour les communications fondées sur le protocole IP, l'identité d'une socket contient l'adresse IP de la machine et le numéro de port employé. En fait, un processus ne devra obligatoirement identifier sa socket que s'il doit être joint par un autre programme. Si le processus doit lui-même contacter un serveur, il lui faut connaître l'identité de l'autre extrémité de la communication, mais l'extrémité locale sera automatiquement identifiée par le noyau.

Pour stocker l'adresse complète d'une socket, on emploie la structure **sockaddr**, définie dans <sys/socket.h> et contenant les membres suivants :

Nom	Type	Signification
sa_family	unsigned short int	Famille de communication
sa_data	char []	Données propres au protocole

En réalité, cette structure est une coquille vide, permettant d'employer un type homogène pour toutes les communications réseau. Pour indiquer véritablement l'identité d'une socket, on utilise une structure dépendant de la famille de communication, puis on emploie une conversion de type (struct sockaddr *) lors des appels-système.

Pour les sockets de la famille AF_INET reposant sur le protocole IP, la structure utilisée est **sockaddr_in**, définie dans <netinet/in.h> :

Nom	Type	Signification
sin_family	short int	Famille de communication AF_INET.
sin_port	unsigned short	Numéro de port, dans l'ordre des octets du réseau.
sin_addr	struct in_addr	Adresse IP de l'interface (dont le membre s_addr est dans l'ordre des octets du réseau).

Nous ne les étudierons pas ici, mais on peut signaler que d'autres familles de protocoles utilisent les structures suivantes :

Structure	Fichier	Famille	Utilisation
sockaddr_ax25	<netax25/ax25.h>	AF_AX25	Radioamateurs
sockaddr_in6	<netinet/in.h>	AF_INET6	IPv6
sockaddr_ipx	<netipx/ipx.h>	AF_IPX	Novell IPX
sockaddr_un	<sys/un.h>	AF_UNIX	Interne système Unix

Dans la plupart des appels-système où on transmettra un pointeur sur une structure `sockaddr_in`, converti en pointeur `struct sockaddr *`, il faudra aussi indiquer la taille de la structure `sockaddr_in`, obtenue grâce à `sizeof()`. En effet, cette taille peut varier suivant les familles de communication.

Pour remplir les champs de la structure `sockaddr_in`, nous emploierons donc la méthode suivante :

1. Mettre à zéro tout le contenu de l'adresse, à l'aide de la fonction `memset()`.
2. Remplir le champ `sin_family` avec `AF_INET`.
3. Remplir le champ `sin_port` avec le membre `s_port` d'une structure `servent` renvoyée par `getservbyname()` ou par `getservbyport()`.
4. Remplir le champ `sin_addr` avec le contenu du membre `h_addr` de la structure `hostent` renvoyée par `gethostbyname()` ou avec le retour de la fonction `inet_aton()`. On convertit explicitement le type `char *` du membre `h_addr` en pointeur sur une structure `in_addr` afin de pouvoir copier son champ `s_addr`, qui est entier. Tout ceci deviendra plus clair dans les exemples à venir.

Nous savons désormais créer une nouvelle socket et préparer la structure décrivant entièrement une adresse complète `AF_INET`. Nous pouvons maintenant examiner comment établir la communication.

Mentionnons auparavant l'existence d'un appel-système nommé **`socketpair()`**, permettant de créer en une seule fois deux sockets, à la manière de `pipe()` :

```
int socketpair (int domaine, int type, int protocole, int sock [2]);
```

Les deux descripteurs de socket sont stockés dans le tableau passé en quatrième argument. À la différence de `pipe()`, les deux sockets sont bidirectionnelles. De plus, cet appel-système n'est disponible que dans le domaine `AF_UNIX`, que nous n'examinerons pas ici. Son intérêt est assez limité car il sert essentiellement à la transmission de descripteurs de fichiers ouverts entre processus. Ceci permet notamment à un serveur privilégié d'ouvrir des fichiers pour le compte de processus non privilégiés qui lui en ont fait la demande, le contrôle des accès se faisant par une procédure interne au serveur, généralement plus complexe que les autorisations gérées par le noyau.

Affectation d'adresse

Nous allons d'abord examiner comment affecter une identité à notre socket. Ceci s'effectue à l'aide de l'appel-système **`bind()`** :

```
int bind (int sock, struct sockaddr * adresse, socklen_t longueur);
```

La socket représentée par le descripteur passé en premier argument est associée à l'adresse passée en seconde position. En réalité, on passe une adresse correspondant au domaine de la socket, par exemple un pointeur sur une structure `sockaddr_in`, converti en pointeur `sockaddr *`. Le noyau connaissant le type de la socket – précisé lors de sa création – assurera à son tour la conversion inverse.

Le dernier argument représente la longueur de l'adresse. Cette valeur est indispensable, car avant d'analyser le type de la socket, le noyau doit copier l'adresse depuis l'espace de l'utilisateur vers son propre espace mémoire, et doit donc connaître la longueur réelle de la structure

en deuxième position. Le type `socklen_t` n'est pas disponible sur tous les Unix. Dans ce cas le troisième argument est un entier `int`.

Le schéma habituel est donc pour une socket en mode connecté :

```
    int
cree_socket_stream (const char * nom_hote,
                    const char * nom_service, const char * nom_proto)
{
    int sock;
    struct sockaddr_in adresse;
    struct hostent *   hostent;
    struct servent *   servent;
    struct protoent *  protoent;

    if ((hostent = gethostbyname(nom_hote)) == NULL) {
        perror("gethostbyname");
        return -1;
    }
    if ((protoent = getprotobyname(nom_proto)) == NULL) {
        perror("getprotobyname");
        return -1;
    }
    if ((servent = getservbyname(nom_service,
                          protoent->p_name)) == NULL) {
        perror("getservbyname");
        return -1;
    }
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return -1;
    }
    memset(& adresse, 0, sizeof (struct sockaddr_in));
    adresse sin_family = AF_INET;
    adresse.sin_port = servent->s_port;
    adresse.sin_addr . s_addr =
             ((struct in_addr *) (hostent->h_addr))->s_addr;
   if (bind(sock, (struct sockaddr *) & adresse,
                sizeof(struct sockaddr_in)) < 0) {
        close(sock);
        perror("bind");
        return -1;
    }
    return sock;
}
```

Notre socket est donc créée et elle possède un nom. Nous allons voir dans les prochains paragraphes comment le serveur peut se mettre à l'écoute en attendant que des processus le contactent.

Lorsqu'on crée une socket située du côté client, il n'est pas indispensable de mentionner explicitement notre identité. Le noyau attribuera de toute façon une adresse correcte lorsqu'on entrera en communication avec un autre processus. Si on désire quand même se servir de `bind()` de ce côté, on peut utiliser comme adresse la constante **`INADDR_ANY`**, définie dans

<netinet/in.h>, qui indique au noyau de choisir l'interface réseau adaptée pour la liaison avec le serveur (en fonction de ses tables de routage). De même, on emploie un numéro de port nul afin de demander au noyau de nous en attribuer un libre.

L'initialisation se fait alors ainsi :

```
    memset(& adresse, 0, sizeof(struct sockaddr_in));
    adresse.sin_family = AF_INET;
    adresse.sin_port = htons(0);
    adresse.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(...
```

Remarquez que l'utilisation de `htons()` n'est pas nécessaire puisque le numéro de port est nul. Il en est de même pour l'utilisation de `htonl()` car la constante `INADDR_ANY` correspond à l'adresse `0.0.0.0`, c'est-à-dire `0x00000000`. Toutefois il est préférable de les employer car on a souvent tendance en programmation réseau à copier-coller des portions de code, et à modifier uniquement les valeurs entre parenthèses. Cette précaution évitera d'avoir à chercher ultérieurement un bogue difficile sur l'ordre des octets.

L'appel-système `bind()` peut échouer en renvoyant –1, avec les conditions d'erreur suivantes :

Nom	Signification
`EBADF, ENOTSOCK`	Le descripteur de socket est invalide.
`EACCESS`	L'adresse demandée ne peut être employée que par un processus ayant la capacité `CAP_NET_BIND_SERVICE`.
`EINVAL`	La socket a déjà une adresse ou elle est déjà connectée, ou encore la longueur indiquée est inexacte.
`EADDRINUSE`	L'adresse est déjà utilisée.

L'erreur `EADDRINUSE` est souvent déclenchée lorsqu'on redémarre un serveur qu'on vient d'arrêter. Nous reviendrons en détail sur ce phénomène lorsque nous étudierons les options des sockets.

Il est possible également de rechercher l'adresse d'une socket. Ceci peut être utile si elle a été identifiée automatiquement par le noyau mais qu'on désire quand même connaître ses caractéristiques, pour les fournir à l'utilisateur par exemple.

L'appel-système **`getsockname()`** renvoie l'adresse d'une socket :

```
int getsockname (int sock, struct sockaddr * adresse,
                 socklen_t * longueur);
```

Le second argument est un pointeur sur une structure correspondant au type de socket employée. Dans le domaine `AF_INET`, on utilise une `sockaddr_in` en convertissant explicitement le pointeur. Le troisième argument doit pointer sur une variable qui contient la longueur de cette structure. Au retour de la fonction, cette variable comprendra le nombre d'octets réellement écrits. L'utilisation de `getsockname()` se fait donc ainsi :

```
    int
affiche_adresse_socket (int sock)
{
    struct sockaddr_in adresse;
    socklen_t          longueur ;
```

```
    longueur = sizeof(struct sockaddr_in);
    if (getsockname(sock, & adresse, & longueur) < 0) {
        perror("getsockname");
        return -1;
    }
    fprintf(stdout, "IP = %s, Port = %u \n",
                inet_ntoa(adresse.sin_addr),
                ntohs(adresse.sin_port));
    return 0;
}
```

De même, lorsqu'une socket est connectée, il est possible d'obtenir des informations sur son correspondant en employant **getpeername()** :

```
int getpeername (int sock, struct sockaddr * adresse,
                 socklen_t * longueur);
```

Cet appel-système fonctionne comme `getsockname()`, mais il nous renseigne sur le correspondant distant. Ceci n'est possible qu'en mode connecté, lorsqu'une communication s'établit de manière organisée entre deux correspondants. Si la socket n'est pas connectée, elle peut avoir une multitude de correspondants successifs, puisqu'on pourra changer d'interlocuteur à chaque écriture, et que tout un chacun pourra lui envoyer des données.

Mode connecté et mode non connecté

Lorsqu'on emploie des sockets fonctionnant en mode non connecté (`SOCK_DGRAM` dans le domaine `AF_INET`), le travail d'initialisation est déjà terminé. Le processus qui attend de recevoir des requêtes de la part d'autres programmes – appelons-le serveur – a identifié sa socket avec `bind()` : elle est accessible de l'extérieur. Du côté des clients, l'identification est établie automatiquement par le noyau. Au contraire, pour une communication connectée, il reste encore du travail à accomplir, tant du côté serveur que du côté client.

La différence essentielle entre les communications en mode connecté et celles en mode non connecté est que cette dernière technique nécessite d'indiquer le destinataire du message à chaque envoi. Au contraire, lorsqu'une connexion est établie, il n'y a plus que deux interlocuteurs face à face, et il n'y a pas d'ambiguïté lors de l'émission ou de la réception d'un message.

Lors d'une communication non connectée, on utilise les appels-système `sendto()` et `recvfrom()`, qui permettent d'envoyer un paquet à destination d'un processus passé en argument ou de recevoir un paquet en récupérant l'adresse de l'émetteur. Nous reviendrons plus loin sur ces primitives. Par contre, en mode connecté un processus peut utiliser `send()` et `recv()`, qui ne précisent pas l'adresse du correspondant, ou même `read()` et `write()` comme avec un tube classique.

Il existe une bonne analogie qu'on retrouve souvent lorsqu'il s'agit de définir la notion de connexion. Pour faire parvenir des informations à un proche, nous pouvons utiliser soit le téléphone, soit le service postal[1].

- Pour communiquer par téléphone nous devons établir la liaison en appelant notre correspondant. Celui-ci décroche et nous nous identifions mutuellement. Le respect de ce protocole

1. On peut aussi envoyer un e-mail, mais cela revient exactement au même principe que l'acheminement postal.

nous permet de dialoguer en mode connecté. Tant que nous n'avons pas raccroché, il est possible d'envoyer des informations sans avoir besoin de préciser le destinataire. Si une interférence se produit, notre correspondant nous demande de répéter la phrase. La communication en mode connecté nous garantit ici que les messages seront bien délivrés à notre interlocuteur, indemnes et dans l'ordre. Le téléphone et ses tonalités caractéristiques (attente, occupé, raccroché), associés à un protocole comportant quelques mots standard « Allô ? », « Pardon, pouvez-vous répéter ? », « Au revoir ! », est un service de communication fiable.

- Lorsqu'on envoie des messages par la poste, nous devons écrire l'adresse du correspondant sur chaque enveloppe. Celui-ci regarde régulièrement dans sa boîte et peut y découvrir simultanément des lettres provenant de plusieurs émetteurs. Pour répondre à notre courrier, il lui faudra écrire une nouvelle lettre et la cacheter en inscrivant notre adresse de retour sur l'enveloppe. Il n'y a pas de connexion établie, l'identité du destinataire doit être indiquée dans chaque message. Rien ne garantit non plus que les messages nous parviendront dans l'ordre ni même qu'ils arriveront un jour. Si une lettre est détrempée par la pluie et quasi illisible, notre correspondant devra nous contacter et nous demander de lui réécrire le message. Ceci n'est pas compris dans le protocole mais se trouve dans la couche applicative de la communication. La poste propose un service de communication non connecté, non fiable.

Dans le domaine `AF_INET`, les communications connectées (TCP) sont fiables, alors que les non-connectées (UDP) ne le sont pas. Ceci n'est toutefois pas une règle absolue, car des méthodes de contrôle et de séquencement peuvent être associées à un protocole sans connexion pour le fiabiliser encore plus. Pour continuer notre analogie, l'envoi de lettres postales en recommandé avec accusé de réception transforme ce service en communication non connectée fiable, mais avec un surcoût sensible.

Nous allons examiner à présent la fin de l'initialisation d'une communication en mode connecté avant d'analyser l'utilisation proprement dite de la socket, pour envoyer ou recevoir des données.

Attente de connexions

Nous allons tout d'abord nous intéresser à un serveur acceptant des connexions. Dans notre domaine `AF_INET`, il s'agira donc d'un serveur TCP. Supposons que nous désirions écrire une application fonctionnant comme un démon, à la manière d'un serveur `FTP`, `TELNET`, `finger`, etc.[1] Notre application doit tout d'abord créer une socket de communication TCP, puis lui associer l'adresse IP et le numéro de port sur lesquels les clients tenteront de la contacter. On peut pour cela utiliser la routine `cree_socket_stream()` que nous avons écrite plus haut. On peut l'améliorer en autorisant le pointeur sur le nom d'hôte à être `NULL`, dans ce cas on emploie une adresse `INADDR_ANY`. Le pointeur sur le nom de service peut aussi être `NULL`, ce qui signifie qu'on demande au noyau de nous attribuer un port en indiquant un numéro zéro :

```
if ((sock = cree_socket_stream(NULL, NULL, "tcp")) < 0)
    exit(EXIT_FAILURE);
```

1. En fait, tous ces serveurs généraux du système fonctionnent un peu différemment en utilisant les services de `inetd`, comme nous le verrons ultérieurement.

Il faudra récupérer l'adresse de notre socket à l'aide de la fonction écrite dans la section précédente pour connaître le numéro de port que les clients devront contacter.

Ensuite, nous devrons indiquer au noyau que nous attendons des connexions sur cette socket. Ceci s'effectue en appelant l'appel-système **`listen()`** :

```
int listen(int sock, int nb_en_attente);
```

Le second argument de cet appel-système demande au noyau de dimensionner une file d'attente des requêtes de connexions. Si une demande de connexion arrive et si le serveur est occupé, elle sera mise dans une file. Si la file est pleine, les nouvelles connexions seront rejetées. En général, on emploie la constante 5 car c'était la limite dans l'implémentation originale des sockets BSD, mais Linux accepte une file contenant jusqu'à 128 connexions en attente.

Attention

N'attachez pas trop d'importance à ce paramètre. Il sert simplement à dimensionner la tolérance du système lorsque plusieurs demandes arrivent simultanément. Ce n'est significatif que pour des serveurs acceptant de très nombreuses requêtes avec un rythme élevé (serveur HTTP par exemple).

L'appel `listen()` n'est pas bloquant, il revient immédiatement. Les erreurs éventuelles concernent uniquement les descripteurs de socket invalides ou les tentatives d'utilisation de `listen()` sur des sockets fonctionnant en mode non connecté.

Une fois que le noyau est informé que le processus désire recevoir des connexions, il faut mettre effectivement le programme en attente. Pour cela on invoque l'appel-système **`accept()`** :

```
int accept (int sock, struct sockaddr * adresse, socklen_t * longueur);
```

Nous allons atteindre ici un point subtil de la programmation réseau, nécessitant un peu d'attention. Lorsqu'on crée un serveur TCP, on veut pouvoir recevoir des connexions en provenance de plusieurs clients. Et de surcroît, de façon simultanée. La socket que nous avons créée est attachée à une adresse IP et à un numéro de port connus des clients. Il s'agit donc d'une socket servant à établir le contact. Toutefois, on ne peut pas se permettre de la monopoliser ensuite pour assurer réellement la communication. Imaginons un serveur FTP qui a installé sa socket sur le port 21 de l'interface réseau de la machine. Un correspondant contacte notre serveur sur ce port et la connexion s'établit. Néanmoins, il est hors de question de bloquer ce port pour transférer toutes les données que notre correspondant désire, car aucun autre client ne pourrait nous contacter pendant ce temps.

Le principe de l'appel-système `accept()` est donc de prendre une demande de connexion en attente – dans la file dimensionnée avec `listen() -`, puis d'ouvrir une nouvelle socket du côté serveur et d'établir la connexion sur celle-ci. La socket originale, celle qui a été passée en argument, reste donc intacte, prête à servir à nouveau pour une demande de connexion. La nouvelle socket créée est renvoyée par `accept()`. Le processus emploiera donc celle-ci pour toute la communication ultérieure.

Les deuxième et troisième arguments de `accept()` fonctionnent comme ceux de `getpeer-name()` et fournissent l'identité du client.

La plupart du temps, un serveur veut pouvoir dialoguer avec plusieurs clients simultanément.

Pour cela le plus simple est d'invoquer `fork()` au retour de `accept()`, et de laisser le processus fils traiter la communication, alors que le père retourne en attente sur `accept()`. Voici un tel schéma :

```
    int
serveur_tcp (void)
{
    int sock_contact;
    int sock_connectee;
    struct sockaddr_in adresse;
    socklen_t longueur;

    sock_contact = cree_socket_stream(NULL, NULL, "tcp");
    if (sock_contact < 0)
        return -1;
    listen(sock_contact, 5);
    fprintf(stdout, "Mon adresse >> ");
    affiche_adresse_socket(sock_contact);
    while (! quitter_le_serveur()) {
        longueur = sizeof(struct sockaddr_in);
        sock_connectee = accept(sock_contact,
                                (struct sockaddr *)& adresse, & longueur);
        if (sock_connectee < 0) {
            perror("accept");
            return -1;
        }
        switch (fork()) {
            case 0 : /* fils */
                close(sock_contact);
                traite_connexion(sock_connectee);
                exit(EXIT_SUCCESS);
            case -1:
                perror("fork");
                return -1;
            default : /* père */
                close(sock_connectee);
        }
    }
    return 0;
}
```

On notera que les processus fils qui se terminent deviennent zombies, car leur processus père ne lit pas les codes de retour. Pour éviter cette situation, on peut éventuellement ajouter une ligne :

```
signal (SIGCHLD, SIG_IGN);
```

ou – mieux – installer un petit gestionnaire pour le signal `SIGCHLD`.

L'exemple suivant va utiliser les routines développées ci-dessus.

La routine `quitter_le_serveur()` renvoie toujours zéro ; nous arrêterons le processus avec Contrôle-C. Le travail du processus fils consiste à déterminer l'adresse de son correspondant

à l'aide de `getpeername()`, à afficher les adresses des deux extrémités de la socket, et à transmettre avec `write()` sa propre adresse au correspondant. Le programme est donc :

exemple_serveur_tcp.c :

```
    #include <stdio.h>
    #include <unistd.h>

    #include <arpa/inet.h>
    #include <netdb.h>
    #include <netinet/in.h>

    #include <sys/types.h>
    #include <sys/socket.h>

    int  cree_socket_stream            (const char * nom_hote,
                                        const char * nom_service,
                                        const char * nom_proto);
    int  affiche_adresse_socket        (int sock);
    int  serveur_tcp                   (void);
    int  quitter_le_serveur            (void);
    void traite_connexion              (int sock);

   [ ... ]

    void
traite_connexion (int sock)
{
    struct sockaddr_in adresse;
    socklen_t          longueur;
    char               buffer[256];

    longueur = sizeof(struct sockaddr_in);
    if (getpeername(sock,(struct sockaddr *)& adresse,& longueur) < 0) {
        perror("getpeername");
        return;
    }
    sprintf(buffer, "IP = %s, Port = %u \n",
            inet_ntoa(adresse.sin_addr),
            ntohs(adresse.sin_port));
    fprintf(stdout, "Connexion : locale ");
    affiche_adresse_socket(sock);
    fprintf(stdout, "          distante %s", buffer);
    write(sock, "Votre adresse : ", 16);
    write(sock, buffer, strlen(buffer));
    close(sock);
}

    int
main (int argc, char * argv[])
{
    return serveur_tcp ();
}
```

Avant de tester ce programme, je voudrais ajouter un mot concernant les fichiers d'en-tête inclus dans les logiciels utilisant les sockets. On l'a vu, ces fichiers définissent un grand nombre de structures dépendant les unes des autres et des macros pour accéder à leurs différents champs. Sous Linux, l'organisation des fichiers est telle que l'ordre d'inclusion n'a pas d'importance. Par contre, sur d'autres systèmes cet ordre est crucial, car la moindre inversion peut déclencher des cascades d'avertissements du compilateur, voire des échecs de compilation. La liste des fichiers d'en-tête inclus dans le programme ci-dessus ainsi que leur ordre représentent ce qui me semble, empiriquement, le plus portable sur d'autres Unix.

Pour nous connecter au serveur, nous utiliserons le programme `telnet` en lui indiquant en argument le numéro de port que le programme affiche au démarrage. Nous allons présenter le serveur sur la partie gauche de l'écran et les clients sur la moitié droite. Les deux premières connexions ont lieu depuis la même machine que le serveur, la troisième depuis une autre station. La première connexion emploie l'adresse *loopback* `127.0.0.1`, alors que les autres passent par l'interface réseau de cette machine `192.1.1.51`. La constante `INADDR_ANY` qui est utilisée comme adresse pour le serveur a pour valeur `0.0.0.0`, ce qui correspond à une écoute sur toutes les interfaces disponibles.

```
$ ./exemple_serveur_tcp
Mon adresse >> IP = 0.0.0.0, Port = 1605
                            $ telnet localhost 1605
                            Trying 127.0.0.1...
                            Connected to localhost.
                            Escape character is '^]'.
Connexion : locale IP = 127.0.0.1, Port = 1605
          distante IP = 127.0.0.1, Port = 1606
                            Votre adresse : IP = 127.0.0.1, Port = 1606
                            Connection closed by foreign host.
                            $

                            $ telnet 192.1.1.51 1605
                            Trying 192.1.1.51...
                            Connected to 192.1.1.51.
                            Escape character is '^]'.
Connexion : locale IP = 192.1.1.51, Port = 1605
          distante IP = 192.1.1.51, Port = 1607
                            Votre adresse : IP = 192.1.1.51, Port = 1607
                            Connection closed by foreign host.
                            $

                            $ telnet 192.1.1.51 1605
                            Trying 192.1.1.51...
                            Connected to 192.1.1.51.
                            Escape character is '^]'.
Connexion : locale IP = 192.1.1.51, Port = 1605
          distante IP = 192.1.1.61, Port = 1025
                            Votre adresse : IP = 192.1.1.61, Port = 1025
                            Connection closed by foreign host.
                            $
    (Contrôle-C)
$
```

Un programme peut avoir besoin d'attendre des connexions simultanément sur plusieurs sockets de contact – par exemple sur plusieurs ports. Or, `accept()` est un appel-système bloquant. Il est donc possible d'utiliser auparavant `select()`, en attendant l'arrivée de données en lecture sur toutes les sockets surveillées. Les données reçues correspondront à une demande de connexion, et on pourra alors appeler `accept()` en sachant qu'on ne restera pas bloqué.

Demander une connexion

Nous allons à présent nous intéresser à la socket située du côté client, en étudiant l'appel-système **`connect()`**.

```
int connect (int sock, struct sockaddr * adresse, socklen_t longueur);
```

Celui-ci fonctionne de manière évidente, en contactant le serveur dont l'adresse est passée en argument et en établissant la connexion sur la socket indiquée. Nous pouvons l'employer pour créer un petit utilitaire, `tcp_2_stdout`, qui se connecte sur un serveur TCP et recopie tout ce qu'il reçoit sur sa sortie standard. Il acceptera les options `–a` et `–p` servant à indiquer respectivement l'adresse et le numéro de port du serveur. Nous créerons donc une routine analysant les arguments en ligne de commande et remplissant une structure `sockaddr_in`, afin de pouvoir la réutiliser dans d'autres programmes.

```
    int
lecture_arguments (int argc, char * argv [],
                   struct sockaddr_in * adresse, char * protocole)
{
    char * liste_options = "a:p:h";
    int    option;
    char * hote  = "localhost";
    char * port = "2000";
    struct hostent * hostent;
    struct servent * servent;
    int    numero;

    while ((option = getopt(argc, argv, liste_options)) != -1) {
        switch (option) {
            case 'a' :
                hote  = optarg;
                break;
            case 'p' :
                port = optarg;
                break;
            case 'h' :
                fprintf(stderr, "Syntaxe : %s [-a adresse] [-p port] \n",
                                    argv[0]);
                return -1;
            default :
                break;
        }
    }
    memset(adresse, 0, sizeof (struct sockaddr_in));
```

```
    if (inet_aton(hote, & (adresse->sin_addr)) == 0) {
        if ((hostent = gethostbyname(hote)) == NULL) {
            fprintf(stderr, "hôte %s inconnu \n", hote);
            return -1;
        }
        adresse->sin_addr.s_addr =
            ((struct in_addr *) (hostent->h_addr))->s_addr;
    }
    if (sscanf(port, "%d", & numero) == 1) {
        adresse->sin_port = htons(numero);
        return 0;
    }
    if ((servent = getservbyname(port, protocole)) == NULL) {
        fprintf(stderr, "Service %s inconnu \n", port);
        return -1;
    }
    adresse->sin_port = servent->s_port;
    return 0;
}
```

Le client TCP devient donc simplement :

tcp_2_stdout.c :

```
    #include <stdio.h>
    #include <unistd.h>

    #include <arpa/inet.h>
    #include <netdb.h>
    #include <netinet/in.h>
    #include <sys/types.h>
    #include <sys/socket.h>

    #define LG_BUFFER 1024

    int lecture_arguments (int argc, char * argv [],
                        struct sockaddr_in * adresse, char * protocole);

    int
main (int argc, char * argv[])
{
    int                 sock;
    struct sockaddr_in adresse;
    char                buffer[LG_BUFFER];
    int                 nb_lus;

    if (lecture_arguments(argc, argv, & adresse, "tcp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
```

```
    if (connect(sock, (struct sockaddr *) & adresse,
                sizeof (struct sockaddr_in)) < 0) {
        perror("connect");
        exit(EXIT_FAILURE);
    }
    setvbuf(stdout, NULL, _IONBF, 0);
    while (1) {
        if ((nb_lus = read(sock, buffer, LG_BUFFER)) == 0)
            break;
        if (nb_lus < 0) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        write(STDOUT_FILENO, buffer, nb_lus);
    }
    return EXIT_SUCCESS;
}
```

L'appel `read()` renvoie zéro lorsque la communication est coupée, et –1 en cas d'erreur, ce qui explique les deux cas traités dans la boucle. On remarque que le buffer de sortie de `stdout` a été supprimé avec `setvbuf()`. Ceci sert principalement lorsque cet utilitaire est employé pour transférer des données binaires, afin qu'elles soient transmises au processus en aval au rythme de leur arrivée depuis le réseau[1].

L'exécution de ce programme donne les mêmes résultats que ce que nous observions avec `telnet` :

```
$ ./exemple_serveur_tcp
Mon adresse >> IP = 0.0.0.0, Port = 1628
                            $ ./tcp_2_stdout -p 1628
Connexion : locale IP = 127.0.0.1, Port = 1628
          distante IP = 127.0.0.1, Port = 1634
                            Votre adresse : IP = 127.0.0.1, Port = 1634
                            $

                            $ ./tcp_2_stdout -p 1700
                            connect: Connexion refusée
                            $
    (Contrôle-C)
$
```

Nous constatons lors de la seconde invocation que l'appel-système `connect()` échoue s'il n'y a pas de serveur sur le port indiqué. On peut très bien utiliser ce programme pour se connecter sur des services du système :

```
$ ./tcp_2_stdout -p daytime
Thu Mar 16 13:26:09 2000
$ ./tcp_2_stdout -p nntp
```

1. Comme je l'ai déjà mentionné dans un autre chapitre, j'ai déjà utilisé professionnellement les utilitaires de transfert entre le réseau et `stdin` ou `stdout` développés ici. Durant des phases de débogage ou de prototypage d'applications aéroportuaires, ils me servaient à transporter et à convertir des données provenant de radars vers des applications de visualisation. Il ne s'agit donc pas d'exemples totalement artificiels.

```
200 Leafnode NNTP Daemon, version 1.9.10 running at venux.ccb.fr
    (Contrôle-C)
$
```

Lorsque nous établissons la connexion avec le démon `nntp`, il nous faut ensuite la couper manuellement avec Contrôle-C car chaque programme est en attente de données provenant de l'autre processus.

Le service `daytime` est implémenté directement dans le démon superserveur `/usr/sbin/inetd`. Il renvoie simplement la date et l'heure du système. Nous pouvons en implémenter une version très facilement en modifiant le programme `exemple_tcp_serveur.c` pour qu'il utilise la routine suivante :

```
    void
traite_connexion (int sock)
{
    char   buffer [256];
    time_t heure;

    heure = time(NULL);
    sprintf(buffer, "%s", ctime(& heure));
    write(sock, buffer, strlen(buffer));
    close(sock);
}
```

Bien entendu, pour un véritable serveur système il faudrait employer le numéro de port approprié, 13 en l'occurrence. On peut vérifier que ce serveur se comporte comme l'original :

```
$ ./exemple_serveur_daytime
Mon adresse >> IP = 0.0.0.0, Port = 1665
                        $ telnet localhost 1665
                        Trying 127.0.0.1...
                        Connected to localhost.
                        Escape character is '^]'.
                        Thu Mar 16 14:27:29 2000
                        Connection closed by foreign host.
                        $

                        $ telnet localhost daytime
                        Trying 127.0.0.1...
                        Connected to localhost.
                        Escape character is '^]'.
                        Thu Mar 16 14:27:32 2000
                        Connection closed by foreign host.
                        $
    (Contrôle-C)
$
```

L'appel-système `connect()` peut aussi être employé sur des sockets UDP. Cela sert à indiquer au noyau que nous désirons dialoguer sur cette socket exclusivement avec le correspondant indiqué en argument. Il sera alors possible d'utiliser directement `read()` et `write()`, ou `recv()` et `send()`, sans avoir besoin de s'occuper à nouveau de l'interlocuteur. Le noyau dirigera toutes nos écritures vers l'adresse et le port indiqués. Parallèlement, tous les messages ne

provenant pas de ce correspondant seront éliminés par le noyau. Ceci permet de filtrer la communication lorsqu'on ne veut pas être dérangé par d'autres processus. L'utilisation de `connect()` sur une socket UDP présente aussi l'avantage de pouvoir mieux gérer les erreurs grâce au protocole ICMP, comme nous le verrons plus bas.

Attention

La connexion d'une socket UDP n'est qu'une opération interne au processus et au noyau. Ce dernier mémorise l'adresse du correspondant préférentiel, mais aucun dialogue réseau n'a lieu. L'interlocuteur n'est aucunement concerné par cette action.

Fermeture d'une socket

Pour refermer une socket, on emploie en général l'appel-système `close()`, qui est adapté à tous les descripteurs de fichiers sous Unix. Cette primitive est automatiquement invoquée lorsqu'un processus se termine. La socket est donc refermée et devient inutilisable.

Toutefois, avec un protocole connecté, il se peut que certaines données n'aient pas encore été transmises ou que l'accusé de réception ne soit pas encore arrivé. Si des données sont toujours en train de circuler sur le réseau, elles peuvent arriver endommagées et le destinataire peut nous demander de les répéter. Le protocole étant fiable, il doit garantir la bonne transmission des données, même si le processus s'est terminé juste après les avoir écrites.

La fermeture immédiate d'une communication TCP n'est donc pas possible. La terminaison est une opération à part entière du protocole, nécessitant des acquittements complets des deux correspondants. Cela signifie que la fermeture d'une socket TCP n'a pas de répercussion immédiate sur les interfaces réseau du noyau. La socket continue d'exister pendant un certain temps, afin de s'assurer que toutes les données restantes ont été transmises. Cette socket est encore visible avec l'utilitaire `netstat`. Si on essaye de réutiliser immédiatement l'adresse en relançant le processus, `bind()` nous renvoie l'erreur `EADDRINUSE`.

Pour en avoir le cœur net, nous pouvons créer un programme qui ouvre une socket serveur TCP, attend une connexion et la referme immédiatement. Ensuite, ce processus va essayer d'invoquer `bind()`, en bouclant jusqu'à ce qu'il réussisse. Il nous affichera alors la durée écoulée.

delai_close.c :

```
#include <errno.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

int lecture_arguments (int argc, char * argv [],
                struct sockaddr_in * adresse, char * protocole);
```

```
    int
main (int argc, char * argv[])
{
    int                 sock;
    struct sockaddr_in adresse;
    time_t              debut;
    time_t              fin;

    if (lecture_arguments(argc, argv, & adresse, "tcp") < 0)
        exit(EXIT_FAILURE);
    adresse . sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (bind(sock, & adresse, sizeof (struct sockaddr_in)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }
    listen(sock, 5);
    close(accept(sock, NULL, 0));
    close(sock);
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    time(& debut);
    while (1) {
        if (bind(sock, (struct sockaddr *) & adresse,
                  sizeof (struct sockaddr_in)) == 0)
            break;
        if (errno != EADDRINUSE) {
            perror("bind 2");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
    time(& fin);
    fprintf(stdout, "Durée de persistance après fermeture : %ld \n",
                                fin - debut);
    return EXIT_SUCCESS;
}
```

Le comptage ne commence que lorsque la première socket est refermée, c'est-à-dire après la fin du `telnet` exécuté sur un autre terminal. On profite du délai pour observer la socket avec `netstat`.

```
$ ./delai_close -a 192.1.1.51 -p 1234
                $ telnet 192.1.1.51 1234
                Trying 192.1.1.51...
                Connected to 192.1.1.51.
                Escape character is '^]'.
                Connection closed by foreign host.
```

```
                $ netstat -t
                Connexions Internet actives (sans serveurs)
                Proto Recv-Q Send-Q  Adr.locale  Adr.dist.    Etat
                tcp    0     0       venux:1234   venux:1671    TIME_WAIT
                $

   (1 minute plus tard)

Durée de persistance après fermeture : 60
$
```

L'état `TIME_WAIT` indiqué par `netstat` correspond à l'attente intégrée dans le protocole TCP. Nous voyons qu'avec ce noyau (Linux 2.2), les sockets TCP persistent pendant une minute après leur fermeture. Nous verrons plus loin comment demander à réutiliser immédiatement une adresse, afin de pouvoir relancer un serveur sans attendre pendant une minute.

Il existe un appel-système nommé **`shutdown()`**, permettant de contrôler plus finement la fin de l'utilisation d'une socket :

```
int shutdown (int sock, int mode);
```

Si on appelle cette routine avec un second argument nul, la socket ne permettra plus de recevoir de données. Si cet argument vaut 1, `shutdown()` interdit l'émission de données sur la socket, et un argument valant 2 est équivalent à `close()`. L'utilisation typique de cette primitive est la suivante :

1. Nous envoyons des données à un serveur TCP. Il s'agit d'un flux d'octets, dont la longueur est arbitraire (donc le serveur ne peut pas déterminer leur fin). Une fois que toutes les informations ont été envoyées, nous fermons le côté écriture de la socket avec `shutdown (sock, 1)`.
2. Les appels-système `read()` invoqués sur le serveur renverront 0 dès que les données auront toutes été envoyées et que nous aurons appelé `shutdown()`. Sachant que toutes les informations sont arrivées, le serveur peut les traiter et nous envoyer la réponse. À la fin de la réponse, le serveur ferme sa socket (et le processus fils se termine).
3. Le client peut lire la réponse car sa socket n'est pas fermée en lecture. Quand toute la réponse aura été reçue, nos appels `read()` renverront 0. On pourra alors fermer la socket totalement.

L'appel-système `shutdown()` ne peut être utilisé que sur une socket connectée.

Recevoir ou envoyer des données

Pour envoyer ou recevoir des données, nous avons jusqu'à présent utilisé `write()` et `read()`, car d'une part nous connaissions déjà ces fonctions et d'autre part nos programmes fonctionnaient en mode connecté avec TCP. Si nous choisissons UDP, il nous faut employer des routines plus générales, permettant d'avoir accès à l'identité de l'interlocuteur. Les fonctions **`recvfrom()`** et **`sendto()`** remplissent ce rôle :

```
int recvfrom (int sock, char * buffer, int taille_buffer, int attributs,
              struct sockaddr * source, socklen_t * taille);
int sendto   (int sock, char * buffer, int taille_buffer, int attributs,
              struct sockaddr * source, socklen_t taille);
```

La seule différence entre ces deux prototypes est le dernier argument. Il s'agit d'un pointeur dans le cas de `recvfrom()` et d'une valeur dans le cas de `sendto()`. Les autres arguments correspondent à la socket employée, au buffer à transmettre ou à remplir, ainsi qu'à sa taille et à des attributs que nous détaillerons plus bas. Les deux derniers arguments définissent une adresse et sa taille.

Dans le cas de `recvfrom()`, la structure `sockaddr` transmise est remplie lors de l'appel-système avec l'adresse de l'émetteur du message lu. Si ce pointeur est `NULL`, il est ignoré.

Dans le cas de `sendto()`, la structure `sockaddr` doit contenir l'adresse du destinataire du message. Si la socket est connectée, ce pointeur peut être `NULL`.

Il existe d'ailleurs deux fonctions plus courtes, **`send()`** et **`recv()`**, équivalentes de `sendto()` et `recvfrom()` avec des pointeurs d'adresse `NULL`.

```
int send (int sock, char * buffer, int taille_buffer, int attributs);
int recv (int sock, char * buffer, int taille_buffer, int attributs);
```

L'appel `send()` ne peut être utilisé qu'avec une socket connectée (en TCP ou en UDP). Il faut en effet que le noyau connaisse l'adresse du correspondant. La primitive `recv()` peut par contre être employée aussi sur une socket non connectée, encore que ce soit inhabituel – si on ne désire pas connaître l'adresse de l'émetteur et si on ne veut donc pas lui répondre.

Les valeurs les plus fréquentes qu'on peut associer par un OU binaire dans le champ `attributs` sont les suivantes :

Nom	Signification
`MSG_DONTROUTE`	Cette option sert surtout à déboguer les communications sur un réseau. Elle permet de négliger les procédures de routage mises en service par le noyau et de diriger directement le message vers l'interface qui correspond au sous-réseau de l'adresse du destinataire. Ne sert qu'avec `sendto()` ou `send()`.
`MSG_OOB`	Le message doit être considéré comme des données TCP hors bande. Elles sont émises avec une priorité supérieure à celle des informations normales. Au niveau du récepteur, elles seront reçues sans passer par une file d'attente. Suivant la configuration de la socket, il peut être nécessaire d'utiliser cette option pour lire les données hors bande.
`MSG_PEEK`	Lire les données désirées sur une socket TCP, sans les extraire de la file d'attente. Ne sert qu'avec `recvfrom()` ou `recv()`.

Il peut exister d'autres options spécifiques au noyau. Comme on le voit, la plupart du temps on n'utilise pas ces valeurs. Il est alors possible d'employer directement `write()` et `read()`, qui sont exactement équivalents à `send()` et `recv()` avec des arguments `attributs` nuls. Ces routines ont l'avantage d'être utilisables sur tout type de descripteurs, depuis les fichiers spéciaux de périphériques aux sockets, en passant par les fichiers normaux, les tubes, etc.

Il faut signaler l'existence de deux appels-système très puissants mais assez complexes, **`sendmsg()`** et **`recvmsg()`**. Ceux-ci utilisent des structures permettant de regrouper plusieurs lectures ou plusieurs écritures, à la manière de `readv()` et de `writev()`. Ils permettent également de transmettre un descripteur de fichier ouvert entre processus. Ces opérations sortent des limites de notre propos, qui est simplement d'expliquer comment faire communiquer des processus répartis.

Lorsqu'un processus tente d'écrire sur une socket n'ayant pas d'interlocuteur, le signal `SIGPIPE` est déclenché. On aura habituellement tout intérêt à l'ignorer, car dans ce cas l'appel-système concerné – `write()`, `recv()`ou `recvfrom()` - renverra une erreur `EPIPE`, plus facile à traiter dans le cours du programme que de manière asynchrone dans un gestionnaire de signaux.

Lorsque la lecture se fait sur une socket connectée dont le correspondant a fermé l'autre extrémité, l'appel-système renvoie une valeur nulle. Si on utilise `select()` sur une socket en lecture et si l'interlocuteur la referme de son côté, cet appel-système signale que des données sont disponibles en lecture. Ceci est dû à l'arrivée d'un caractère `EOF`. La lecture suivante renverra zéro octet.

Pour observer un peu `recvfrom()` et `sendto()` sur des sockets UDP, nous allons créer deux utilitaires : `udp_2_stdout`, qui permet de recevoir des données sur un port UDP et de les transmettre sur sa sortie standard, et en parallèle `stdin_2_udp`, qui sert à diriger des données vers un serveur en écoute.

stdin_2_udp.c :

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>

    #include <arpa/inet.h>
    #include <netdb.h>
    #include <netinet/in.h>
    #include <sys/types.h>
    #include <sys/socket.h>

    #define LG_BUFFER    1024

    int lecture_arguments (int argc, char * argv [],
            struct sockaddr_in * adresse, char * protocole);

    int
main (int argc, char * argv[])
{
    int                sock;
    struct sockaddr_in adresse;
    char               buffer[LG_BUFFER];
    int                nb_lus;

    if (lecture_arguments(argc, argv, & adresse, "udp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    while (1) {
        if ((nb_lus = read(STDIN_FILENO, buffer, LG_BUFFER)) == 0)
            break;
```

```
        if (nb_lus < 0) {
            perror("read");
            break;
        }
        sendto(sock, buffer, nb_lus, 0,
            (struct sockaddr *) & adresse, sizeof (struct sockaddr_in));
    }
    return EXIT_SUCCESS;
}
```

udp_2_stdout.c :

```
    int
main (int argc, char * argv[])
{
    int                sock;
    struct sockaddr_in adresse;
    char               buffer[LG_BUFFER];
    int                nb_lus;

    if (lecture_arguments(argc, argv, & adresse, "udp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (bind(sock, & adresse, sizeof(struct sockaddr_in)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }
    setvbuf(stdout, NULL, _IONBF, 0);
    while (1) {
        if ((nb_lus = recv(sock, buffer, LG_BUFFER, 0)) == 0)
            break;
        if (nb_lus < 0) {
            perror("read");
            break;
        }
        write(STDOUT_FILENO, buffer, nb_lus);
    }
    return EXIT_SUCCESS;
}
```

On notera que l'adresse et le numéro de port indiqués en arguments doivent dans les deux cas correspondre au processus récepteur. Nous pouvons essayer ces applications en lançant simultanément deux `stdin_2_udp` et un `udp_2_stdout`. L'enchaînement n'est pas très facile à représenter :

```
$ ./udp_2_stdout -a 192.1.1.51 -p 1234
            $ ./stdin_2_udp -a 192.1.1.51 -p 1234
                        $ ./stdin_2_udp -a 192.1.1.51 -p 1234
```

```
            Première chaîne depuis station 1
Première chaîne depuis station 1
                        Première chaîne depuis station 2
Première chaîne depuis station 2
            Deuxième chaîne, depuis station 1
Deuxième chaîne, depuis station 1
            (Contrôle-C)
            $
                        Deuxième chaîne, depuis station 2
Deuxième chaîne, depuis station 2
                        (Contrôle-C)
                        $
(Contrôle-C)
$
```

Nous n'avons pas utilisé jusqu'à présent l'aspect bidirectionnel des sockets. Nous allons maintenant le mettre en œuvre en créant un serveur TCP qui reçoit des chaînes de caractères émises par un client, les traite et les renvoie au même client qui les affiche. Pour avoir quelque chose à faire avec les chaînes, nous allons à nouveau créer une application particulièrement utile : un serveur d'anagrammes…

Le serveur est une variation sur `exemple_serveur_tcp.c`, dans lequel nous modifions la routine de traitement des connexions ainsi :

```
    void
traite_connexion (int sock)
{
    char buffer[256];
    int  longueur;

    while (1) {
        longueur = read(sock, buffer, 256);
        if (longueur < 0) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        if (longueur == 0)
            break;
        buffer[longueur] = '\0';
        strfry(buffer);
        write(sock, buffer, longueur);
    }
    close(sock);
}
```

De son côté, le client est construit à partir de `stdin_2_tcp.c`, en modifiant la routine principale :

```
    int
main (int argc, char * argv [])
{
    int                sock;
```

```
    struct sockaddr_in adresse;

    char                buffer [LG_BUFFER];
    int                 nb_lus;

    if (lecture_arguments(argc, argv, & adresse, "tcp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (connect(sock, & adresse, sizeof (struct sockaddr_in)) < 0) {
        perror("connect");
        exit(EXIT_FAILURE);
    }
    while (1) {
        if (fgets(buffer, 256, stdin) == NULL)
            break;
        if (buffer[strlen(buffer) - 1] == '\n')
            buffer[strlen(buffer) - 1] = '\0';
        if (write(sock, buffer, strlen(buffer)) < 0) {
            perror("write");
            break;
        }
        if ((nb_lus = read(sock, buffer, LG_BUFFER)) == 0)
            break;
        if (nb_lus < 0) {
            perror("read");
            break;
        }
        fprintf(stdout, "%s\n", buffer);
    }
    return EXIT_SUCCESS;
}
```

L'exécution se déroule comme prévu :

```
$ ./exemple_serveur_anagramme
Mon adresse >> IP = 0.0.0.0, Port = 1693
                $ ./exemple_client_anagramme -a 192.1.1.51 -p 1693
                anagramme
                managrmea
                linux
                lixnu
                fin
                inf
                  (Contrôle-C)
                $
  (Contrôle-C)
$
```

Accès aux options des sockets

Il existe de multiples options configurables concernant les sockets. La plupart ne sont utiles qu'à des fins de débogage des protocoles réseau ou pour des applications très spécifiques. Pourtant, certaines d'entre elles sont couramment employées. Il existe deux appels-système, **`getsockopt()`** et **`setsockopt()`**, permettant de lire l'état d'une option de configuration ou de la modifier.

```
int getsockopt (int sock, int niveau, int option,
                void * valeur, socklen_t * longueur);
int setsockopt (int sock, int niveau, int option,
                const void * valeur, socklen_t longueur);
```

Le premier argument de ces routines est l'identificateur de la socket concernée. Le second correspond au niveau auquel s'applique l'option. Ce niveau représente en fait la couche de protocole correspondant à l'option désirée. Pour nous, il s'agira uniquement des valeurs `SOL_SOCKET` indiquant qu'il s'agit de la socket elle-même, `IPPROTO_IP` correspondant à la couche réseau IP, ou `IPPROTO_TCP` pour la couche de transport TCP. Pour chaque option présentée ci-dessous, nous préciserons le niveau d'application.

Le troisième argument représente l'option elle-même. En quatrième argument on trouve un pointeur sur une variable contenant la valeur associée à l'option. Avec `getsockopt()`, cette variable sera remplie, avec `setsockopt()` elle sera lue. Enfin, on trouve en dernier argument la longueur de la variable employée pour stocker la valeur. Cette longueur doit dans tous les cas être initialisée avant l'appel, même si elle peut être modifiée par `getsockopt()`. Toutes les options présentées ci-dessous utilisent une valeur de type `int`, considérée comme vraie si elle est non nulle, à l'exception de `SO_LINGER` qui emploie une structure `linger`.

Les options les plus courantes pour le niveau `SOL_SOCKET` sont les suivantes :

Option	Signification
`SO_BROADCAST`	Autorisation de diffusion de messages *broadcast* sur une socket UDP. Nous décrirons ce mécanisme plus bas.
`SO_BSDCOMPAT`	Lorsqu'une socket UDP est connectée, une tentative d'écriture vers un port où personne n'écoute renverra une erreur ICMP. Ceci est également vrai sous Linux avec une socket UDP non connectée. Cette option – spécifique à Linux – force le noyau à adopter un comportement BSD, en n'envoyant pas cette erreur sur les socket non connectées.
`SO_DEBUG`	Activation des procédures de débogage dans les couches réseau du noyau.
`SO_DONTROUTE`	Contournement des procédures de routage, les messages étant directement dirigés vers l'interface correspondant à la partie sous-réseau de l'adresse du destinataire.
`SO_ERROR`	Uniquement avec `getsockopt()` : renvoie la valeur d'erreur correspondant à la socket (les erreurs sont identiques à celles de `errno`).
`SO_KEEPALIVE`	Activation d'un envoi périodique de messages sur une socket TCP connectée pour tester sa validité. Si le correspondant ne les acquitte pas, la communication est rompue. Ceci n'est généralement pas intéressant car le délai entre deux messages est de l'ordre de plusieurs heures.
`SO_LINGER`	Activation d'un délai de latence lors d'un appel `close()` s'il reste des données non émises. Rarement utile.

Option	Signification
SO_OOBINLINE	Autorisation pour que les données hors bande arrivant sur une socket soient placées dans le flux normal de lecture, sans nécessiter d'option particulière de recv(). Ceci permet de transmettre des messages avec des priorités supérieures à d'autres.
SO_RCVBUF et SO_SNDBUF	Indique la taille du buffer de réception ou d'émission.
SO_RCVLOWAT et SO_SNDLOWAT	Ces valeurs correspondent à des seuils inférieurs dans les buffers de réception et d'émission, qui déclenchent – lorsqu'ils sont dépassés – une réponse positive de select() pour la socket.
SO_RCVTIMEO et SO_SNDTIMEO	Uniquement avec getsockopt(), ces valeurs représentent un délai maximal en réception et en émission.
SO_REUSEADDR	Autorisation de réutiliser une adresse déjà affectée. Cette option est présentée ci-dessous.
SO_TYPE	Uniquement avec getsockopt(), renvoie la valeur correspondant au type de socket SOCK_STREAM, SOCK_DGRAM, SOCK_RAW…

Les options qui intéressent en général le programmeur applicatifs sont donc essentiellement SO_BROADCAST et SO_REUSEADDR, et parfois SO_BSDCOMPAT.

L'option **SO_REUSEADDR** permet notamment de relancer immédiatement un serveur TCP qu'on vient d'arrêter sans obtenir l'erreur EADDRINUSE lors du bind(). On insère l'appel setsockopt() avant le bind():

```
int autorisation;

autorisation = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
                & autorisation, sizeof(int));
```

Notez bien que cette option est très utile, et que l'on peut considérer que tous les serveurs TCP/IP devraient l'activer.

L'option **SO_BROADCAST** permet d'effectuer de la diffusion globale, ce qui consiste à envoyer un message UDP en direction de tout un sous-réseau. L'ensemble des stations ayant une adresse IP dans ce sous-réseau recevra le paquet de données et le fera remonter jusqu'à la couche UDP. Si une application est en écoute sur le bon numéro de port, elle recevra les informations. Ce mécanisme permet d'arroser tout un ensemble de machines avec des données.

L'adresse de diffusion correspondant à un sous-réseau est obtenue en remplissant tout l'espace réservé pour les adresses des stations par des 1 binaires. Ainsi, sur un sous-réseau de classe C 192.1.1., la diffusion *broadcast* s'obtient en envoyant des données à l'adresse 192.1.1.255. Pour être sûr de ne pas effectuer cette opération de manière fortuite, il faut l'indiquer explicitement dans la configuration de la socket :

```
int autorisation;

sock = socket(AF_INET, SOCK_DGRAM, 0);
autorisation = 1;
setsockopt(sock, SOL_SOCKET, SO_BROADCAST,
            & autorisation, sizeof(autorisation);
```

Ensuite, on peut envoyer des messages à l'adresse de diffusion. La réception se fait de manière transparente, il suffit de lire les données sur le port adéquat.

La diffusion *broadcast* est un mécanisme très utile dans certaines situations, quand toutes les machines du même sous-réseau assurent une tâche similaire (affichage ou calculs en parallèle) avec des données identiques. Cependant, même les stations non intéressées par les données sont obligées de les faire remonter jusqu'à la couche UDP, où elles seront rejetées à cause de leur numéro de port. Ceci implique une surcharge de travail parfois importante.

Afin d'affiner le filtrage, il existe un autre mécanisme de diffusion, employant des adresses *multicast* qui sont traitées directement au niveau de l'interface réseau et de la couche IP.

Le principe de diffusion *multicast* consiste à obliger une application désireuse de recevoir les données à s'inscrire explicitement dans le groupe de diffusion. Cette inscription se fait au niveau de la couche réseau. Ainsi une machine dans laquelle aucun processus n'est intéressé par ces informations n'a pas besoin de les laisser remonter dans ses couches IP et UDP.

Les adresses de groupes *multicast* se trouvent dans l'intervalle IP `224.0.0.0` à `239.255.255.255`. Pour envoyer des données à tout un groupe de diffusion, il suffit donc d'écrire dans une socket UDP dirigée sur l'une de ces adresses.

Pour gérer son inscription et recevoir ainsi les informations, un processus doit renseigner une structure **`ip_mreq`**, définie ainsi :

Nom	Type	Signification
`imr_multiaddr`	`struct in_addr`	Adresse du groupe de diffusion qu'on désire rejoindre.
`imr_interface`	`struct in_addr`	Interface réseau à employer pour joindre le groupe. En général, on utilise `INADDR_ANY`.

Pour joindre un groupe, il faut utiliser l'option **`IP_ADD_MEMBERSHIP`** du niveau `IPPROTO_IP` de la socket (ce niveau d'options est présenté dans un tableau plus bas). Cette option prend en argument une structure `ip_mreq`. À partir de ce moment, les données à destination de l'adresse indiquée dans `ip_mreq.imr_multiaddr` remonteront jusqu'à la couche UDP de la machine réceptrice. Ensuite, elles seront disponibles sur le numéro de port qui leur est attribué. Le processus récepteur doit donc invoquer également `bind()`, pour préciser le port sur lequel il écoute.

> Sous Linux, on peut invoquer `bind()` avant ou après l'inscription dans le groupe multicast ; cela n'a pas d'importance. Sous d'autres Unix et sous Windows, il faut que le `bind()` soit appelé en premier.

Le programme `udp_2_stdout.c` peut être modifié pour recevoir des données *multicast*.

exemple_reception_multicast.c :

```
    int
main (int argc, char * argv[])
{
    int                 sock;
    struct ip_mreq      requete_multicast;
    struct sockaddr_in adresse;
    char                buffer[LG_BUFFER];
```

```
    int                 nb_lus;

    if (lecture_arguments(argc, argv, & adresse, "udp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    requete_multicast.imr_multiaddr.s_addr =
                                adresse.sin_addr.s_addr;
    requete_multicast.imr_interface.s_addr = htons(INADDR_ANY);

    adresse.sin_addr.s_addr = htons(INADDR_ANY);
    if (bind(sock, & adresse, sizeof(struct sockaddr_in)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            & requete_multicast, sizeof(struct ip_mreq)) < 0) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    setvbuf(stdout, NULL, _IONBF, 0);
    while (1) {
        if ((nb_lus = recv(sock, buffer, LG_BUFFER, 0)) == 0)
            break;
        if (nb_lus < 0) {
            perror("read");
            break;
        }
        write(STDOUT_FILENO, buffer, nb_lus);
    }
    return EXIT_SUCCESS;
}
```

Pour que la réception fonctionne, il faut que le système sache que les paquets dirigés vers l'adresse *multicast* du groupe choisi doivent être transférés à la couche IP, sinon ils seront rejetés dès la couche réseau. Il faut donc utiliser `/sbin/route`. Supposons que nous voulions utiliser le groupe *multicast* `224.0.0.0`. Nous allons configurer la réception, puis passer en écoute, une autre machine enverra des données *multicast*.

```
$ su
Password:
# /sbin/route add 224.0.0.0 dev eth0
# exit
exit
$ ./exemple_reception_multicast -a 224.0.0.0 -p 1234
```

```
                                $ ./stdin_2_udp -a 224.0.0.0 -p 1234
                                Chaîne émise en multicast
Chaîne émise en multicast
                                    (Contrôle-C)
                                $
   (Contrôle-C)
$
```

Les options du niveau `IPPROTO_IP` qui nous concernent sont celles qui ont trait à la diffusion *multicast*, les autres étant à un niveau trop bas dans le protocole :

Options	Signification
`IP_ADD_MEMBERSHIP` et `IP_DROP_MEMBERSHIP`	Demande l'inscription ou le désistement d'une socket UDP dans un groupe *multicast*. La valeur est représentée sous forme d'une structure `ip_mreq` décrite plus haut.
`IP_MULTICAST_IF`	Permet de préciser l'interface réseau à utiliser pour recevoir les données. La valeur est une structure `in_addr`.
`IP_MULTICAST_LOOP`	Activation ou non d'un écho *via* l'interface *loopback* des messages *multicast* émis si l'émetteur est membre du groupe de réception.
`IP_MULTICAST_TTL`	Configuration du champ *Time-To-Live* du paquet IP diffusé en *multicast*. Il s'agit d'une valeur entière indiquant approximativement le nombre de routeurs que le paquet peut franchir avant d'être détruit. Normalement, cette valeur vaut 1 pour limiter la portée des messages au même réseau physique.

Les options du niveau `IPPROTO_TCP` sont essentiellement les suivantes :

Option	Signification
`TCP_MAXSEG`	Configuration de la taille maximale des segments de données transmis par le protocole. Doit être inférieur à la valeur de MTU de l'interface réseau (en général 1500 pour les cartes Ethernet et les liaisons PPP).
`TCP_NODELAY`	Cette option empêche TCP de mettre en attente – quelques centièmes de secondes – les petits paquets de données pour essayer de les regrouper en un seul gros paquet, afin de limiter la surcharge due aux en-têtes TCP. Ceci n'est utile que si de toutes petites quantités de données doivent être traitées très rapidement (des mouvements de souris par exemple).

Seule l'option `TCP_NODELAY` peut parfois avoir une utilité dans les applications courantes.

Programmation d'un démon ou utilisation de inetd

Lorsqu'un serveur de données TCP a atteint un niveau de maturité fonctionnelle suffisant pour présenter un intérêt global au niveau du système et du réseau (par exemple un serveur d'anagrammes), il est souvent intéressant de le faire fonctionner en tant que démon.

Un démon est un processus tournant en arrière-plan sur le système, sans terminal de contrôle. En général, les démons sont démarrés lors de l'initialisation du système, et on les laisse s'exécuter jusqu'à l'arrêt de la machine. Pour transformer un serveur classique en démon, il faut respecter certaines règles :

1. Tout d'abord le démon doit passer en arrière-plan. Pour cela on utilise :
```
if (fork() != 0)
    exit (EXIT_SUCCESS);
```
2. Le démon ne doit bloquer aucune partition du système – sauf s'il s'agit de ses propres répertoires comme `/var/spool/lpd` pour le démon `lpd`. Aussi il faudra en général remonter à la racine du système de fichiers :
```
chdir("/");
```
3. Le processus doit créer une nouvelle session et s'assurer qu'il n'a pas de terminal de contrôle. Nous avons déjà observé ceci dans le chapitre 2 :
```
setsid();
if (fork() != 0)
    exit(EXIT_SUCCESS);
```
4. Finalement, le démon doit fermer tous les descripteurs de fichiers que le shell aurait pu lui transmettre. Une méthode courante est d'utiliser :
```
for (i = 0; i < OPEN_MAX; i ++)
    close(i);
```

Naturellement, le démon ne pourra plus afficher de message sur `stderr`, il lui faudra employer le mécanisme `syslog()` que nous avons étudié dans le chapitre 26.

Le programme `exemple_demon_anagramme.c` est une réplique de `exemple_serveur_anagramme.c` dans lequel nous avons remplacé toutes les occurrences de

```
perror("xxx");
```

par

```
syslog(LOG_ERR, "xxx : %m");
```

et

```
fprintf(stdout, "IP = %s, port = %u \n", ...);
```

par

```
syslog(LOG_INFO, "IP = %s, port = %u", ...);
```

La fonction principale est devenue :

```
    int
main (int argc, char * argv[])
{
    int i;

    chdir("/");
    if (fork() != 0)
        exit(EXIT_SUCCESS);
    setsid();
    if (fork() != 0)
        exit(EXIT_SUCCESS);
    for (i = 0; i < OPEN_MAX; i ++)
        close(i);
```

```
    serveur_tcp();
    return EXIT_SUCCESS;
}
```

Le programme se comporte tout à fait comme un démon classique :

```
$ ./exemple_demon_anagramme
$ ps aux | grep "[d]emon"
ccb   1979  0.0  0.4  1232  516 ?    S    22:53   0:00 ./exemple_demon_a
$ tail /var/log/messages
  [...]
Mar 17 22:53:52 venux exemple_demon_anagramme: IP = 0.0.0.0, Port = 1059
$ ./exemple_client_anagramme -p 1059
linux
uxlni
  (Contrôle-C)
$ killall exemple_demon_anagramme
$ ps aux | grep "[d]emon"
$
```

En fait, ce programme gagnerait à employer un numéro de port figé, inscrit dans `/etc/services`, plutôt que de nous obliger à regarder le fichier de messages de `syslog()` pour le trouver.

Une alternative à la programmation d'un démon est l'emploi du superserveur réseau `inetd`. Ce démon lit au démarrage sa configuration dans `/etc/inetd.conf` et assure toute la gestion de l'aspect serveur TCP. Lorsqu'une connexion a été établie, il invoque directement l'utilitaire demandé, en ayant redirigé – grâce à `dup()` – son entrée et sa sortie standard vers la socket obtenue.

Sur la plupart des distributions Linux – mais pas toutes – `inetd` a été remplacé par `xinetd`, son successeur apportant de nombreuses options supplémentaires, notamment en ce qui concerne la sécurité du système.

Notre serveur d'anagrammes peut alors être réécrit tout simplement ainsi :

exemple_inet.anagramme.c :

```
    #define _GNU_SOURCE
    #include <ctype.h>
    #include <stdio.h>
    #include <string.h>
    #include <unistd.h>

    int
main (void)
{
    char chaine[256];
    int  n;

    while (1) {
        if ((n = read(STDIN_FILENO, chaine, 256)) <= 1)
            break;
        while (isspace(chaine[n - 1]))
            n --;
        chaine[n] = '\0';
```

```
        strfry(chaine);
        write(STDOUT_FILENO, chaine, n);
    }
    return EXIT_SUCCESS;
}
```

Il nous faut alors ajouter un port dédié dans `/etc/services` et une ligne de lancement dans `/etc/inetd.conf`. On se reportera aux pages de manuel `inetd(8)` et `inetd.conf(8)` pour plus de détails sur la syntaxe.

```
$ su
Password:
# cp exemple_inet_anagramme /usr/local/bin/
# vi /etc/services
   (édition et ajout de la ligne)
anagramme 2000/tcp
# vi /etc/inetd.conf
   (édition et ajout de la ligne)
anagramme stream tcp nowait root /usr/local/bin/exemple_inet_anagramme
# killall -HUP inetd
# exit
$
```

On peut à présent tester le serveur aussi bien avec le logiciel client dédié qu'avec `telnet`.

```
$ ./exemple_client_anagramme -p 2000
linux
lxniu
client
neilct
$ telnet localhost 2000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
unix
niux
serveur
sueerrv
Connection closed by foreign host.
$
```

Conclusion

Nous avons étudié dans ce chapitre les rudiments de la programmation réseau en TCP/IP et UDP/IP, en soulignant bien la puissance du concept de socket, puisque ce mécanisme simple et qui hérite logiquement des méthodes classiques de communication entre processus permet de donner une dimension nouvelle aux applications en leur offrant de dialoguer avec des stations réparties dans le monde entier.

Nous avons limité notre présentation aux informations les plus utiles pour un programmeur applicatif. Bien entendu, il existe encore de nombreux points qui n'ont pas été abordés. La référence essentielle dans ce domaine est [STEVENS 1990] *UNIX Network Programming*.

On pourra trouver de précieux renseignements dans la Faq *Unix Socket Faq for Network Programming*, postée dans le groupe Usenet `comp.unix.programmer`.

On se reportera également aux documents Linux *NET-3-HOWTO* et *Multicast-HOWTO*.

Les RFC restent la source ultime – mais peu digeste – de référence en ce qui concerne le réseau. Nous en avons indiqué quelques-unes dans le chapitre précédent.

33

Gestion du terminal

La gestion des terminaux, sous Linux du moins, est souvent et principalement motivée par l'un des trois principes suivants :

- Les entrées-sorties habituelles ne permettent pas de capturer des caractères au vol sans que l'utilisateur n'appuie sur la touche Entrée. Dans certaines applications, on peut avoir besoin de modifier le mode du terminal pour adopter ce type de comportement.
- Lorsqu'on veut proposer une connexion à distance, à la manière des démons `telnetd` ou `rlogind`, il est souvent nécessaire d'utiliser des fonctionnalités particulières offertes par le noyau, par le biais des pseudo-terminaux.
- Sur un PC sous Linux, les lignes d'entrée-sortie série ne sont généralement pas utilisées pour connecter de véritables terminaux mais plutôt pour brancher des équipements divers – modems, imprimantes, matériel personnel –, et l'initialisation de ces ports RS-232 emploie les méthodes de configuration des terminaux.

Nous allons donc étudier les fonctions de manipulation des terminaux avec ces trois optiques successives.

Définition des terminaux

La notion de terminal Unix a été introduite dans les systèmes où une unité centrale accueillait les connexions en provenance d'une ou plusieurs dizaines de terminaux travaillant en mode texte. Depuis quelques années ces ensembles ont pratiquement disparu, remplacés par des stations de travail intégrant l'unité de calcul et un terminal graphique haute résolution, connectés par réseau Ethernet afin de partager grâce au protocole NFS les ressources disque.

Cette évolution est encore plus marquée avec les PC sous Linux, sur lesquels il est bien rare qu'une machine dispose de plus d'un ou deux ports RS-232, et encore plus rare que ceux-ci servent à connecter des terminaux en mode texte, sauf à titre de curiosité purement expérimentale. Toutefois les mécanismes de configuration sont toujours disponibles et servent à gérer les consoles virtuelles, les pseudo-terminaux ou des liaisons RS-232 avec divers équipements.

Chaque terminal qu'on pouvait connecter sur une unité centrale offrait des possibilités spécifiques, et sa configuration différait toujours quelque peu des autres modèles. Ces paramètres concernent par exemple les codes servant à effacer l'écran, à déplacer le curseur ou à changer la couleur du texte. Une liste imposante de terminaux différents avec leurs caractéristiques est donc disponible dans le fichier `/etc/termcap` des systèmes Unix.

Le noyau gère le terminal de manière à offrir une interface homogène aux processus lorsqu'ils y accéderont avec les appels-système `read()` et `write()`. Pour configurer l'état d'un terminal, le noyau propose une série de paramètres regroupés dans une structure `termios`, qu'on nomme *discipline de ligne*. Nous reviendrons sur son contenu plus bas.

La discipline de ligne s'intercale donc entre le terminal et le processus. Le noyau incorpore aussi deux buffers pour conserver les touches appuyées afin qu'elles soient lues par le programme et pour enregistrer les caractères envoyés, jusqu'à ce que le terminal soit prêt à les afficher.

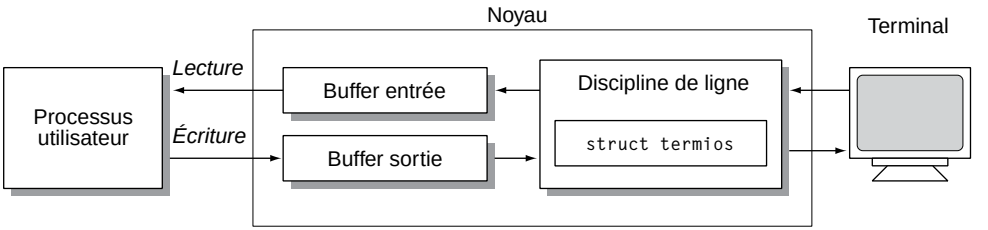


Figure 33.1

Organisation de la configuration du terminal

Les méthodes de configuration des terminaux étaient à l'origine fondées sur l'appel-système `ioctl()`, qui offre une interface de bas niveau avec les périphériques d'entrée-sortie. Ce mécanisme est complexe et peu portable. D'autres fonctions ont donc été définies par Posix, bien qu'il reste encore quelques extensions BSD ou Système V disponibles sous Linux.

Un terminal peut être utilisé dans deux modes nommés traditionnellement *canonique* et *non canonique* :

- En mode canonique, les saisies sont transmises au processus ligne par ligne. Ceci signifie que tant que l'utilisateur n'a pas appuyé sur la touche Entrée, le pilote de terminal n'envoie rien au processus qui effectue un `read()`. Cette configuration est celle établie par défaut sur les terminaux sous Unix, nous en avons déjà parlé à de multiples reprises dans ce livre.
- En mode non canonique ou brut (*raw*), le terminal transmet immédiatement les caractères sur lesquels on appuie. Un `read()` ne nous retournera donc pas nécessairement des lignes entières.

Chacun de ces modes présente des avantages :

Le mode canonique permet de laisser le pilote de terminal gérer entièrement les corrections, effacement, etc. Lors d'une saisie avec `fgets()`, l'utilisateur pourra revenir en arrière avec la touche d'effacement et corriger sa ligne. Seule la pression sur la touche Entrée validera vraiment la saisie.

Le mode non canonique permet de construire des applications d'édition de texte plus dynamiques, puisqu'on pourra gérer un curseur avec les touches fléchées, les sauts de page, etc. Le processus peut aussi capturer des caractères au vol dans des applications où les saisies de l'utilisateur se déroulent en parallèle avec d'autres tâches.

Une ligne, en mode canonique, est soumise à une longueur maximale de **MAX_CANON** caractères, définie dans `<limits.h>`. Cette valeur, 255 sous Linux, est souvent utilisée pour dimensionner des chaînes de caractères avant d'appeler `fgets()`.

Configuration d'un terminal

La structure **termios** comporte cinq membres définis par SUSv3 :

Nom	Type	Signification
`c_iflag`	`tcflag_t`	Attributs définissant le mode d'entrée depuis le terminal
`c_oflag`	`tcflag_t`	Attributs concernant le mode de sortie vers le terminal
`c_cflag`	`tcflag_t`	Attributs permettant le contrôle du terminal
`c_lflag`	`tcflag_t`	Attributs locaux pour le terminal
`c_cc`	`cc_t[]`	Variables concernant le terminal. Essentiellement des affectations de touches

Nous allons examiner en détail les attributs un peu plus loin. Indiquons tout de suite que la répartition des attributs dans les différents membres n'a rien d'intuitif et que certaines fonctionnalités, comme la gestion des signaux, nécessitent d'agir sur plusieurs attributs, en l'occurrence `c_iflag` et `c_cflag`.

La configuration du terminal s'effectue à l'aide des fonctions **tcgetattr()** et **tcsetattr()**, qui permettent de lire ou de fixer la valeur de la structure `termios`.

```
int tcgetattr (int terminal, struct termios * configuration);
int tcsetattr (int terminal, int option, struct termios * configuration);
```

Attention

Les structures `termios` comportant des champs avec un grand nombre d'attributs, il est important de lire l'état du terminal avec `tcgetattr()` avant de modifier les membres voulus, puis de la réécrire avec `tcsetattr()`.

Le premier argument de ces fonctions doit être un descripteur de terminal, par exemple `STDIN_FILENO`. La routine `tcsetattr()` utilise une option qui précise à quel moment les modifications éventuelles doivent avoir lieu. Il peut s'agir de l'une des constantes suivantes :

Nom	Signification
`TCSANOW`	Les modifications sont appliquées immédiatement.
`TCSADRAIN`	Les modifications prendront effet lorsque tous les caractères en attente d'affichage auront été lus par le terminal.
`TCSAFLUSH`	Comme avec `TCSADRAIN`, on attend que le contenu du buffer de sortie ait été transmis, mais de plus le buffer d'entrée sera automatiquement purgé.

Tant qu'aucune information n'a été transmise, juste après l'ouverture du terminal, on pourra donc employer `TCSANOW` pour le configurer, mais ensuite on utilisera `TCSAFLUSH` ou `TCSADRAIN` en fonction de l'importance qu'on accorde aux données en attente de lecture.

Le retour de `tcsetattr()` est problématique, car cette fonction n'échoue que si aucune modification n'a pu être apportée à la structure `termios` originale. Pour vérifier que tout s'est bien passé, il est donc nécessaire de relire à nouveau la configuration avec `tcgetattr()` et de comparer le résultat avec les options demandées. L'attitude défensive, voire paranoïaque, à adopter est la suivante :

1. Lecture de la structure `termios` avec `tcgetattr()`.
2. Modification des membres de la structure.
3. Écriture des nouveaux attributs avec `tcsetattr()`.
4. Lecture de la configuration réellement acceptée avec `tcgetattr()`.
5. Comparaison entre les modifications demandées et celles qui ont été obtenues.

L'utilisation de `TCSAFLUSH` permet d'implémenter la fonctionnalité que de trop nombreux débutants espèrent obtenir en invoquant `fflush(stdin)`, qui n'a aucune signification. Il suffit de recopier la configuration du terminal sans la modifier. En voici un exemple :

exemple_flush.c :

```
    #include <stdio.h>
    #include <termios.h>
    #include <unistd.h>

    int
main (void)
{
    struct termios terminal;
    int i;

    fprintf(stdout, "FLUSH dans 5 secondes \n");
    sleep(5);
    fprintf(stdout, "FLUSH !\n");
    if (tcgetattr(STDIN_FILENO, & terminal) == 0)
        tcsetattr(STDIN_FILENO, TCSAFLUSH, & terminal);
    while ((i = fgetc(stdin)) != EOF)
        fprintf(stdout, "%02X ", i);
    return EXIT_SUCCESS;
}
```

Les caractères saisis pendant les cinq secondes d'attente du programme sont purgés lors du `tcsetattr()` suivant :

```
$ ./exemple_flush
FLUSH dans 5 secondes
Ceci sera oublié

FLUSH !
   (Contrôle-D)
$
```

Il existe une fonction **tcflush()** qui assure le même rôle, en permettant de vider au choix le buffer d'entrée ou celui de sortie :

```
int tcflush (int terminal, int buffer);
```

Le second argument peut prendre l'une des valeurs suivantes :

Nom	Signification
TCIFLUSH	Purge du buffer d'entrée, comme nous l'avons fait avec tcsetattr().
TCOFLUSH	Purge du buffer de sortie.
TCIOFLUSH	Purge des deux buffers.

Si on désire s'assurer que toutes les données écrites ont bien été traitées par le terminal, on peut utiliser la fonction **tcdrain()**. Celle-ci est bloquante tant que le buffer de sortie n'est pas vide.

```
int tcdrain (int descripteur);
```

Attention

Si on se sert des fonctions de la bibliothèque stdio, il faut penser à employer fflush(stdout) avant d'utiliser tcdrain(), sinon des données pourraient encore se trouver dans la mémoire tampon interne à la bibliothèque.

On peut aussi vouloir bloquer temporairement l'entrée ou la sortie sur un terminal. Ceci est surtout utilisé dans des applications de communication, afin d'assurer un contrôle de flux entre les deux extrémités. La fonction **tcflow()** permet de gérer ainsi les deux canaux indépendamment :

```
int tcflow (int terminal, int blocage);
```

Le second argument peut prendre l'une des valeurs suivantes :

Nom	Signification
TCIOFF	On envoie au terminal un caractère STOP (voir plus loin). Il va donc arrêter d'émettre des données afin que nous puissions traiter le contenu du buffer d'entrée.
TCION	On envoie un caractère START au terminal. Il peut reprendre l'envoi de données.
TCOOFF	On interdit temporairement l'émission de données (suite à la réception d'une requête du terminal). Les tentatives d'écriture seront bloquantes.
TCOON	On débloque l'émission de données. Les écritures pourront reprendre.

La routine tcflow() peut être employée dans un gestionnaire de signaux. Ceci permet d'implémenter un contrôle de flux. Le gestionnaire peut être invoqué en réception de données, par exemple avec l'une des méthodes de traitement asynchrone étudiées dans le chapitre 30. Lors de l'arrivée de certaines conditions (caractères spéciaux), le gestionnaire peut ainsi bloquer le buffer de sortie. Si des tentatives d'écriture ont lieu dans le corps du programme, il restera bloqué, sinon il se déroulera normalement. Lorsque la condition de déblocage se

présentera, le gestionnaire pourra libérer le buffer de sortie et relancer ainsi l'exécution normale du processus.

Notons l'existence d'une dernière fonction de cette famille, **tcsendbreak()**, qui permet d'envoyer un caractère *Break* sur la ligne série du terminal :

```
int tcsendbreak (int terminal, int duree);
```

Un signal *Break* n'est pas vraiment un caractère mais plutôt une succession de bits à zéro pendant une durée supérieure au temps d'émission d'un caractère. Ce signal est donc reconnu par le périphérique connecté sur la liaison série comme une pseudo-erreur réclamant son attention. L'utilisation dépend ensuite du protocole de haut niveau choisi ; on peut l'employer pour synchroniser le dialogue par exemple. Le second argument représente la durée, mais l'unité n'est pas définie par SUSv3. Aussi est-il conseillé d'employer une valeur nulle, ce qui correspond, de manière portable, à un caractère *Break* qui dure entre un quart et une demi-seconde.

Nous allons à présent examiner les valeurs possibles pour les membres de la structure `termios`. Certaines constantes sont définies par SUSv3, d'autres sont des extensions BSD ou Système V. La liste des attributs est un peu fastidieuse, mais il est utile de bien voir les différents éléments entrant en jeu, principalement en ce qui concerne la table `c_cc[]`.

Membre c_iflag de la structure termios

Ce champ est composé d'un OU binaire entre différents arguments, permettant de définir la discipline de ligne pour les caractères provenant du terminal. Sauf indication contraire, ces arguments sont définis par `EXIT_SUCCESS`.

Nom	Signification
`BRKINT` `IGNBRK`	Lorsqu'un caractère *Break* arrive, il est ignoré si `IGNBRK` est présent. Sinon, si `BRKINT` est actif, il déclenchera le signal `SIGINT` sur le processus en avant-plan. Si aucun de ces attributs n'est présent, le caractère *Break* sera lu comme un zéro « `\0` » présentant une erreur de cadrage (voir `PARMRK` ci-dessous).
`ICRNL` `IGNCR`	Si `IGNCR` est activé, les caractères de retour chariot sont ignorés en entrée. Si `ICRNL` est présent, ils sont transformés en saut de ligne. Sinon, ils sont lus comme des caractères « `\r` ».
`IGNPAR` `PARMRK`	Si `IGNPAR` est actif, un caractère présentant une erreur de cadrage (trop long ou trop court) ou de parité est ignoré. Si `PARMRK` est présent, le caractère est préfixé du code d'erreur `0xFF`, `0x00` (un caractère `0xFF` valide est alors indiqué sous forme `0xFF 0xFF`). Sinon, le caractère est simplement lu comme un zéro « `\0` ».
`IMAXBEL`	Extension BSD et Système V non définis par SUSv3. Avec cet attribut, le système déclenchera un bip sur le terminal lorsque le buffer d'entrée sera plein. La taille du buffer d'entrée est de `MAX_INPUT` caractères. Cette valeur (255 sous Linux) est définie dans `<limits.h>`.
`INLCR`	Avec cet attribut, un caractère « nouvelle ligne » reçu en entrée sera converti en retour chariot. Sinon, il est lu comme le caractère « `\n` » habituel.
`INPCK`	Cet attribut active la vérification de la parité des caractères reçus.
`ISTRIP`	Si `ISTRIP` est actif, les caractères reçus en entrée seront tronqués pour tenir sur 7 bits. Le huitième bit est purement et simplement supprimé.
`IUCLC`	Extension Système V non définis par SUSv3. Un caractère majuscule reçu en entrée est transformé en minuscule. Les caractères sur 8 bits sont ramenés sur 7 bits comme avec `ISTRIP`.

Nom	Signification
IXON	Lorsque cet attribut est actif, le terminal assure le contrôle de flux. Les caractères STOP et START définis dans la table c_cc[] que nous verrons plus bas servent à bloquer ou à débloquer l'envoi de données par le processus. Sinon, ces caractères parviennent indemnes au programme.
IXOFF	Avec cet attribut, l'ordinateur gère le contrôle de flux en entrée en envoyant les caractères STOP et START au terminal, en fonction du remplissage du buffer d'entrée.

Membre c_oflag de la structure termios

OPOST sert à autoriser l'action des autres attributs du champ c_oflag.

Noms	Signification
BSDLY BS0, BS1	BSDLY correspond à un masque de bits recouvrant les attributs de configuration du délai de retour en arrière *Backspace*. Ce masque doit donc être effacé du champ c_oflag avant d'y inscrire l'une des deux valeurs possibles de délai BS0 ou BS1.
CRDLY CR0, CR1, CR2, CR3	CRDLY est le masque de bits correspondant aux délais de retour chariot. Les quatre valeurs possibles vont de CR0 à CR3.
FFDLY FF0, FF1	FFDLY est le masque de délai de saut de page, avec deux valeurs possibles, FF0 et FF1.
NLDLY NL0, NL1	Ces attributs représentent le masque et les deux valeurs pour le changement de ligne.
OCRNL ONLCR	Avec OCRNL, les caractères de retour chariot « \r » seront remplacés par des caractères « \n » de nouvelle ligne. Avec ONLCR, la conversion inverse a lieu.
OFDEL	Extension non décrite par SUSv3. Le caractère de remplissage, utilisé pour temporiser les sorties, sera le caractère *DEL* 0x7F si cet attribut est actif. Sinon, il s'agira du caractère nul.
OFILL	Cet attribut demande au pilote de périphérique d'utiliser le remplissage avec un caractère nul ou *DEL* pour gérer les délais de temporisation. Sinon, le pilote attendra simplement sans rien envoyer.
OLCUC	Extension non décrite par SUSv3. Les caractères minuscules seront transformés en majuscules en sortie avec cet attribut. Les caractères hors de la table Ascii sont tronqués à 7 bits.
ONLRET	Avec l'attribut ONLRET, on suppose que le caractère « \n » effectue également le travail du retour chariot « \r » en sortie.
ONOCR	Si cet attribut est actif, aucun caractère « \r » ne sera envoyé en première position d'une ligne.
TABDLY TAB0, TAB1 TAB2, TAB3 XTABS	Masque et valeurs pour le délai de tabulation horizontale. La valeur XTABS demande au pilote de périphérique de remplacer les tabulations par des espaces, en plaçant les taquets toutes les huit colonnes.
VTDLY VT0, VT1	Masque et valeurs de délai pour la tabulation verticale.

Membre c_cflag de la structure termios

Dans ce champ, on trouve des informations concernant la liaison série entre l'unité centrale et le terminal. Dans le cas où le terminal n'est pas réellement connecté par une liaison série (consoles virtuelles, pseudo-terminaux), ce champ est ignoré.

Il existe quelques extensions BSD sur d'autres systèmes, mais les attributs utilisés sous Linux sont tous définis par SUSv3, à l'exception du contrôle de flux CTS/RTS.

Noms	Signification
CLOCAL	Cet attribut sert à inhiber l'utilisation des lignes de contrôle du modem.
CREAD	Lorsque cet attribut est actif, la réception de données est possible.
CRTSCTS	Extension non définie par SUSv3. Le contrôle de flux avec les signaux RTS/CTS est activé par cet attribut.
CSIZE CS5, CS6, CS7, CS8	CSIZE représente le masque binaire recouvrant les bits utilisés pour définir la longueur des caractères transmis. On emploie les constantes CS5 à CS8 pour fixer la taille du caractère. Nous en verrons un exemple plus tard.
CSTOPB	Lorsque cet attribut est actif, les caractères seront délimités par deux bits d'arrêt. Sinon, ils n'auront qu'un seul bit d'arrêt.
HUPCL	Si cet attribut est activé lorsque le descripteur est refermé par le dernier processus qui le tenait ouvert, les signaux de contrôle sont abaissés et le modem est alors raccroché.
PARENB PARODD	L'attribut PARENB active l'utilisation de la parité. Lorsque PARODD est présent, il s'agit d'une parité impaire, sinon c'est une parité paire.

Membre c_lflag de la structure termios

Ce champ contient la configuration de la partie locale du pilote de périphérique, recouvrant surtout les notions d'écho des caractères saisis et la gestion des signaux provenant du terminal.

Nom	Signification
ECHO	Cet attribut représente l'écho des caractères saisis.
ECHOCTL	Extension BSD et Système V non définis par SUSv3: écho des caractères de contrôle (inférieurs à 0x1F) sous forme de lettres préfixées du symbole ^ comme dans la table Ascii présentée en annexe.
ECHOE	En mode canonique, le caractère ERASE efface la lettre précédente, et WERASE efface le mot précédent.
ECHOK	En mode canonique, le caractère KILL efface toute la ligne.
ECHONL	En mode canonique, le caractère « \n » est renvoyé en écho même si l'attribut ECHO n'est pas activé.
FLUSHO	Extension BSD et Système V, cet attribut est basculé à la réception du caractère DISCARD. Il indique que le buffer de sortie doit être purgé.
ICANON	Attribut permettant de passer du mode brut au mode canonique.
IEXTEN	Activation du mode étendu dans le traitement des entrées depuis le terminal. Le comportement de certains caractères de contrôle que nous examinerons ci-dessous dépend de cet attribut.
ISIG	Activation des signaux dus aux caractères INTR, QUIT, SUSP ou DSUSP.
NOFLSH	Cet attribut indique que les buffers d'entrée et de sortie ne doivent pas être vidés lorsque les signaux SIGINT ou SIGQUIT sont déclenchés par le terminal. Ceci s'applique aussi au buffer d'entrée seul, avec le signal SIGSUSP.
PENDIN	Avec cet attribut, qui est une extension BSD et Système V, la frappe d'une touche oblige à réafficher tous les caractères qui ne sont pas encore lus.
TOSTOP	Cet attribut indique qu'un processus en arrière-plan qui tente d'écrire sur son terminal de contrôle recevra le signal SIGTTOU. Nous avons observé ce phénomène dans le chapitre 6.

Membre c_cc[ ] de la structure termios

Finalement, ce membre contient une table de NCCS valeurs, représentant pour la plupart des caractères ayant des fonctions particulières. Il existe des constantes pour accéder aux différents indices dans cette table.

Voyons tout d'abord les caractères spéciaux définis par c_cc[].

Nom	Défaut	Signification
DISCARD	^o	Ce caractère peut être configuré dans c_cc[VDISCARD]. Si la saisie étendue IEXTEN est configurée dans c_lflag, ce caractère désactive l'écriture jusqu'à l'arrivée d'un second DISCARD.
EOF	^d	Ce caractère peut être configuré dans c_cc[VEOF]. En mode canonique, il indique la fin d'une saisie. L'appel-système read() se termine en renvoyant zéro octet.
EOL		Ce caractère peut être configuré dans c_cc[VEOL]. Il s'agit d'un second caractère de saut de ligne en mode canonique.
EOL2		Ce caractère peut être configuré dans c_cc[VEOL2]. Il s'agit encore d'un caractère supplémentaire indiquant la fin de ligne.
ERASE	^h	Ce caractère peut être configuré dans c_cc[VERASE]. Il permet, en mode canonique, d'effacer le dernier caractère saisi. Si le début de ligne est atteint, il n'a pas d'effet.
INTR	^c	Ce caractère peut être configuré dans c_cc[VINTR]. Si l'attribut ISIG est présent dans c_lflag, le signal SIGINT est envoyé aux processus du groupe en avant-plan au moment de sa réception.
KILL	^u	Ce caractère peut être configuré dans c_cc[VKILL]. En mode canonique, il sert à effacer la ligne en cours.
LNEXT	^v	Ce caractère peut être configuré dans c_cc[VLNEXT]. En mode canonique, avec l'attribut IEXTEN dans c_lflag, il sert à lire la valeur du caractère suivant qui est appuyé sans l'interpréter. Par exemple la séquence ^v ^u affiche le caractère ^u sans effacer la ligne.
QUIT	^\	Ce caractère peut être configuré dans c_cc[VQUIT]. Si l'attribut ISIG est actif, le signal SIGQUIT est envoyé à tous les processus du groupe en avant-plan, ce qui par défaut les termine avec un fichier core.
REPRINT	^r	Ce caractère peut être configuré dans c_cc[VREPRINT]. En mode canonique, si IEXTEN est activé dans c_lflag, ce caractère demande le réaffichage de la ligne complète.
START	^q	Ce caractère peut être configuré dans c_cc[VSTART]. Si l'attribut IXON est configuré dans c_iflag, ce caractère sert à redémarrer la lecture si elle était bloquée. Si l'attribut IXOFF est présent dans c_iflag, le pilote envoie automatiquement ce caractère vers le terminal lorsque le buffer d'entrée se trouve à nouveau disponible.
STOP	^s	Ce caractère peut être configuré dans c_cc[VSTOP]. Il s'agit du caractère complémentaire de START, servant à bloquer les écritures s'il est reçu et si l'attribut IXON est actif. De même, il est envoyé automatiquement au terminal lorsque le buffer d'entrée est plein, si IXOFF est présent.
SUSP	^z	Ce caractère peut être configuré dans c_cc[VSUSP]. Si l'attribut ISIG est présent dans c_lflag et si le contrôle des jobs est actif, le signal SIGTSTP est envoyé aux processus du groupe en avant-plan, ce qui suspend les processus sans les terminer.
WERASE	^w	Ce caractère peut être configuré dans c_cc[VWERASE]. En mode canonique, avec l'attribut IEXTEN, ce caractère sert à effacer le dernier mot en reculant jusqu'à rencontrer un caractère non blanc, puis en effaçant tous les caractères jusqu'à un caractère blanc.

Il existe deux caractères spéciaux qui ne peuvent pas être modifiés et qui n'ont donc pas réellement d'emplacement dans la table `c_cc[]` :

Nom	Défaut	Signification
`CR`	`\r`	Ce caractère peut être converti par le pilote de terminal en mode canonique, en fonction des attributs `ICRNL` ou `IGNCR` du membre `c_iflag`.
`NL`	`\n`	En mode canonique, ce caractère sert à délimiter les lignes. En fonction de l'attribut `INLCR` du champ c_iflag, il peut être converti en `\r`.

Les caractères `CR`, `EOL`, `EOL2` et `NL` sont les seuls qui peuvent être renvoyés par un appel `read()` dans le mode canonique. Tous les autres caractères de contrôle sont gérés entièrement par le pilote de terminal. Bien entendu, en mode brut, le pilote transmet tous les caractères au processus lecteur.

En plus de ces caractères, la table `c_cc []` contient deux valeurs numériques :

Indice	Signification
`VTIME`	En mode non canonique, durée maximale d'attente d'un appel `read()`. Nous détaillerons ceci plus bas.
`VMIN`	En mode non canonique, nombre minimal de caractères à renvoyer pour qu'un appel `read()` se termine avant le délai ci-dessus.

Pour tester facilement toutes les options de configuration offertes par la structure `termios`, on peut utiliser le programme `/bin/stty`. Celui-ci travaille sur le descripteur de son entrée standard et affiche l'état de la structure `termios`, ou nous permet de la modifier. Pour consulter tous les attributs `termios`, il faut utiliser `stty -a`. Même lorsqu'il s'agit de transformer une option, `stty` travaille sur le descripteur de son entrée standard, ce qui nous conduit à écrire des choses comme

```
$ stty -crtscts < /dev/ttyS0
```

pour supprimer le contrôle de flux matériel sur le premier port série.

En ce qui concerne le terminal « classique », `stty` permet de modifier pratiquement toutes les options `termios` en les faisant figurer sur la ligne de commande, éventuellement précédées d'un « `-` » pour les désactiver. Il existe une option `sane` qui permet de rétablir le terminal dans un état normal. Pour faire des expériences avec `stty`, il faut éviter que le shell n'interfère, aussi avons-nous deux possibilités :

- Utiliser un shell minimal, qui ne gère pas le clavier de manière trop complète, à la manière de `/bin/ash`.
- Écrire une petite application bouclant en lecture et en exécution de ce type :

```
    while (fgets(chaine, MAX_CANON, stdin) != NULL)
        system(chaine);
```

On se reportera à la page de manuel de `stty` pour voir toutes ces options.

```
$ ash
% stty -a
speed 9600 baud; rows 0; columns 0; line = 0;
```

```
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl
ixon -ixoff -iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0
bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop
-echoprt echoctl echoke
% stty istrip
% (Pression sur éèç) : ihg
ihg: not found
% stty -onlcr
% ls
  Makefile  exemple_pty  exemple_raw.c
                                    exemple_flush  exemple_pty.c
exemple_serie_1
               exemple_flush.c  exemple_raw     exemple_serie_1.c
                                                            % stty sane
% ls
Makefile         exemple_pty    exemple_raw.c
exemple_flush    exemple_pty.c  exemple_serie_1
exemple_flush.c  exemple_raw    exemple_serie_1.c
% exit
$
```

Basculement du terminal en mode brut

Lorsqu'un terminal est initialisé, il se trouve par défaut en mode canonique. Il est possible de passer en mode brut en lisant la structure `termios` à l'aide de `tcgetattr()`, puis de modifier la structure avant de la réécrire dans le terminal avec `tcsetattr()`.

Le passage en mode brut correspond aux opérations suivantes :

- Effacement des attributs `BRKINT`, `IGNBRK`, `ICRNL`, `IGNCR`, `PARMRK`, `INLCR`, `ISTRIP`, `IXON` de `c_iflag`, `OPOST` de `c_oflag`, `ECHO`, `ECHONL`, `ICANON`, `IEXTEN`, `ISIG` de `c_lflag` et `PARENB` de `c_cflag`.
- Configuration d'une taille de caractère `CS8` dans `c_cflag` (après en avoir effacé le masque `CSIZE`).
- Configuration de `c_cc[VTIME] = 0` et `c_cc[VMIN] = 1`.

Pour simplifier cette opération, on utilise donc une fonction de bibliothèque nommée **`cfmakeraw()`**, qui réalise ces étapes.

```
void cfmakeraw (struct termios * configuration);
```

Un programme qui passe un terminal en mode brut doit s'assurer de restituer le mode canonique avant de se terminer. Pour cela, il suffit de garder une copie de la structure `termios`.

Lorsqu'on demande une lecture sur un descripteur de terminal en mode brut, le comportement est dicté par les variables `c_cc[VMIN]` et `c_cc[VTIME]`. La première correspond au nombre minimal de caractères qui doivent être lus pour que l'appel `read()` revienne avant que

son délai ne soit écoulé. Ce délai est indiqué en dixièmes de seconde dans la seconde variable. Les cas possibles sont donc les suivants :

c_cc[VMIN]	c_cc[VTIME]	Réaction à un read()
0	0	L'appel-système `read()` revient immédiatement. Si des caractères sont disponibles, ils sont renvoyés tout de suite.
> 0	0	L'appel `read()` ne revient que lorsque `MIN` caractères au moins sont disponibles. Sinon, il peut bloquer indéfiniment.
0	> 0	L'appel-système ne durera qu'au plus `TIME` dixièmes de seconde. Si des caractères arrivent entre-temps, il se termine, sinon il renvoie 0 à l'expiration du délai.
> 0	> 0	L'appel-système `read()` ne reviendra que si `MIN` caractères sont reçus ou si le délai de `TIME` dixièmes de seconde entre deux caractères est écoulé. Le délai est réinitialisé à l'arrivée de chaque nouveau caractère. Attention, le délai n'est pris en compte qu'après l'arrivée du premier caractère.

Naturellement, dans toutes ces situations `read()` peut se terminer avant l'instant indiqué s'il a lu le nombre de caractères demandés en argument ou si un signal l'interrompt.

En général, on utilisera :

- Soit `c_cc[VMIN] = 1` et `c_cc[VTIME] = 0` pour avoir une lecture brute bloquante. Ceci permet par exemple d'implémenter un éditeur de texte, il s'agit de la configuration mise en place par `cfmakeraw()`.
- Soit `c_cc[VMIN] = 0` et `c_cc[VTIME] = 0` pour une lecture non bloquante, afin de laisser le programme se dérouler en parallèle, comme dans un jeu.

Dans le programme suivant nous allons basculer en mode brut avec lecture non bloquante. En attendant que l'utilisateur saisisse un caractère, le programme fait tourner une barre sur la gauche de la ligne de saisie. À chaque pression sur une touche, le code hexadécimal du caractère correspondant est affiché. Le programme se termine en appuyant sur « `q` ».

exemple_raw.c :

```
    #include <stdio.h>
    #include <termios.h>
    #include <unistd.h>

    struct termios sauvegarde;

    int    initialisation_clavier (int fd);
    int    restauration_clavier   (int fd);

    int
main (void)
{
    char c = 0;
    int  i = 0;
    char * chaine = "-\\|/";

    initialisation_clavier(STDIN_FILENO);
    while (1) {
```

```
        if (read(STDIN_FILENO, & c, 1) == 1)
            if (c == 'q')
                break;
        fprintf(stdout, "\r%c (%02X)", chaine[i], c);
        fflush(stdout);
        if (chaine[++ i] == '\0')
            i = 0;
        usleep(100000);
    }
    restauration_clavier(STDIN_FILENO);
    return EXIT_SUCCESS;
}

    int
initialisation_clavier (int fd)
{
    struct termios configuration;
    if (tcgetattr(fd, & configuration) != 0)
        return -1;
    memcpy(& sauvegarde, & configuration, sizeof (struct termios));
    cfmakeraw(& configuration);
    configuration.c_cc[VMIN] = 0;
    if (tcsetattr(fd, TCSANOW, & configuration) != 0)
        return -1;
    return 0;
}

    int
restauration_clavier (int fd)
{
    tcsetattr(fd, TCSANOW, & sauvegarde);
    return 0;
}
```

Nous ne pouvons évidemment pas montrer d'exemple d'exécution ici, puisque l'intérêt de ce programme est son comportement dynamique.

On voit qu'avec ces fonctionnalités, il est possible de définir des routines à la manière des `kbhit()`, `getch()` ou `getche()` du monde Dos. Les touches de fonction (`F1`, `F2`…), les flèches de déplacement ou les touches de contrôle (`Inser`, `Suppr`, etc.) renvoient des codes composés par plusieurs octets, généralement préfixés par `0x1B` (`ESC`). La gestion de ces touches est spécifique au terminal et est peu portable.

Pour utiliser toutes les possibilités d'action d'un clavier de manière portable, il faudra se tourner vers les fonctionnalités *curses*, dont l'implémentation sous Linux est assurée par la bibliothèque *ncurses*. Cette bibliothèque donne accès à toutes les manipulations de texte en plein écran. Il existe de nombreuses applications utilisant *ncurses*, comme `gdb`, `xwpe`, `vim`, `telnet`, `mc`, etc.

Avant d'essayer de modifier la configuration d'un descripteur, il convient de vérifier s'il s'agit bien d'un terminal. Pour cela on utilise la fonction **`isatty()`**, déclarée dans `<unistd.h>` :

```
int isatty (int descripteur);
```

Cette fonction renvoie une valeur non nulle si le descripteur est bien un terminal. Dans ce cas il est possible d'employer la fonction **ttyname()** pour retrouver le nom du périphérique associé, par exemple /dev/ttyS1.

```
char * ttyname (int descripteur);
```

La chaîne renvoyée se trouve dans une zone de mémoire statique. La bibliothèque GlibC propose également une version réentrante **ttyname_r()** :

```
int ttyname_r (int descripteur, char * buffer, int longueur);
```

Enfin, il existe une routine nommée **ctermid()**, permettant de récupérer le nom du terminal de contrôle du processus. Il ne s'agit pas toujours du véritable nom du périphérique mais généralement de /dev/tty. Cette chaîne est toutefois valide pour ouvrir un descripteur avec open().

```
char * ctermid (char * chaine);
```

Si le pointeur passé en argument est NULL, ctermid() renvoie une chaîne de caractères allouée dans une zone de mémoire statique. Sinon, il stocke le nom dans le buffer transmis, lequel doit contenir au moins **L_ctermid** caractères.

Voyons quelques possibilités d'utilisation de ces routines.

exemple_isatty.c :

```
    #include <stdio.h>
    #include <unistd.h>

    int
main (void)
{
    if (isatty(STDIN_FILENO))
        fprintf(stdout, "stdin : %s\n", ttyname (STDIN_FILENO));
    else
        fprintf(stdout, "stdin : Pas un terminal ! \n");
    fprintf(stdout, "Terminal de contrôle : %s\n", ctermid (NULL));
    return 0;
}
```

Voici un exemple d'exécution depuis une connexion réseau par telnet :

```
$ ./exemple_isatty
stdin : /dev/pts/4
Terminal de contrôle : /dev/tty
$ ./exemple_isatty < exemple_isatty.c
stdin : Pas un terminal !
Terminal de contrôle : /dev/tty
$
```

Par contre, depuis une console virtuelle Linux :

```
$ ./exemple_isatty
stdin : /dev/tty2
Terminal de contrôle : /dev/tty
$
```

L'intérêt essentiel de trouver le nom du terminal de contrôle est de pouvoir effectuer une saisie même si l'entrée standard a été redirigée dans un fichier. La fonction **getpass()**, proposée par la bibliothèque GlibC permet de lire un mot de passe depuis le terminal de contrôle du processus. Cette routine est par exemple employée par /bin/su.

```
char * getpass (const char * invite);
```

Le message transmis en argument est affiché avant de lire le mot de passe. Naturellement, la bibliothèque C supprime l'écho sur le terminal durant la saisie.

Pour connaître ou modifier l'identité du groupe de processus se trouvant en avant-plan sur un terminal, on peut utiliser **tcgetpgrp()** et **tcsetpgrp()** :

```
pid_t tcgetpgrp (int descripteur);
int   tcsetpgrp (int descripteur, pid_t groupe);
```

Nous avons examiné les notions de groupe de processus en avant-plan et de terminal de contrôle d'une session dans le chapitre 2.

Connexion à distance sur une socket

Lorsqu'on désire permettre à un utilisateur de se connecter à distance sur une application, il suffit de mettre en place un serveur TCP, comme nous l'avons vu dans le chapitre précédent. On peut écrire les messages à afficher sur le terminal de l'utilisateur dans la socket avec write() et lire ses réponses avec read(). L'utilisateur pourra se connecter avec telnet, par exemple pour administrer le logiciel à distance.

Le programme suivant va tenter d'implémenter tout ceci en utilisant dup2() pour rediriger l'entrée et la sortie standard vers la socket de communication, puis en appelant system() sur les chaînes de caractères saisies.

exemple_socket.c :

```
    #include <limits.h>
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <unistd.h>

    #include <arpa/inet.h>
    #include <netdb.h>
    #include <netinet/in.h>
    #include <sys/types.h>
    #include <sys/socket.h>

    void
gestionnaire (int numero)
{
    exit(EXIT_SUCCESS);
}
```

```
    void
traite_connexion (int fd)
{
    char chaine [MAX_CANON];
    char * fin;
    FILE * fp;

    if ((fp = fdopen(fd, "r+")) == NULL) {
        perror("fdopen");
        exit(EXIT_FAILURE);
    }
    if (! isatty(fd)) {
        strcpy(chaine, "Vous n'êtes pas connecté sur un terminal ! \n");
        write(fd, chaine, strlen(chaine));
    }
    dup2(fd, STDIN_FILENO);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);
    while (fgets(chaine, MAX_CANON, fp) != NULL) {
        if ((fin = strpbrk(chaine, "\n\r")) != NULL)
            fin[0] = '\0';
        if (strcasecmp(chaine, "fin") == 0) {
            kill(getppid(), SIGINT);
            exit(EXIT_SUCCESS);
        }
        system(chaine);
    }
    exit(EXIT_SUCCESS);
}

    int
main (void)
{
    int                sock;
    int                sock_2;
    struct sockaddr_in adresse;
    socklen_t          longueur;

    if (signal(SIGINT, gestionnaire) != 0) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    signal(SIGCHLD, SIG_IGN);
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    memset(& adresse, 0, sizeof(struct sockaddr));
    adresse.sin_family = AF_INET;
    adresse.sin_addr.s_addr = htonl(INADDR_ANY);
    adresse.sin_port = 0;
```

```
    if (bind(sock, (struct sockaddr *) & adresse,
                        sizeof(adresse)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }
    longueur = sizeof(struct sockaddr_in);
    if (getsockname (sock, (struct sockaddr *) & adresse,
                         & longueur) < 0) {
        perror("getsockname");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Mon adresse : IP = %s, Port = %u \n",
                        inet_ntoa(adresse.sin_addr),
                        ntohs(adresse.sin_port));
    listen(sock, 5);
    while (1) {
        longueur = sizeof(struct sockaddr_in);
        sock_2 = accept(sock, & adresse, & longueur);
        if (sock_2 < 0)
            continue;
        switch (fork()) {
            case 0 : /* fils */
                close(sock);
                traite_connexion(sock_2);
                exit(EXIT_SUCCESS);
            default :
                close(sock_2);
                break;
        }
    }
    close(sock);
    return EXIT_SUCCESS;
}
```

Au début, ce système semble fonctionner correctement, mais on s'aperçoit assez vite de ses limites, notamment si on essaye d'invoquer une commande gérant l'écran en entier, comme l'éditeur `vi`, ou si on veut saisir un mot de passe caché :

```
$ ./exemple_socket
Mon adresse : IP = 0.0.0.0, Port = 1122
                                $ telnet localhost 1122
                                Trying 127.0.0.1...
                                Connected to localhost.
                                Escape character is '^]'.
                                Vous n'êtes pas connecté sur un terminal !
                                ls
                                Makefile
                                exemple_flush
                                exemple_flush.c
                                [...]
                                exemple_socket
                                exemple_socket.c
```

```
                                    echo "un message"
                                    un message
                                    su
                                    standard in must be a tty
                                    login
                                    login: nom
                                    Password: visible
                                    Login incorrect
                                    login: (Contrôle-D)
                                    Login incorrect
                                    login: (Contrôle-D)
                                    Login incorrect
                                    pwd
                                    /home/ccb/Doc/ProgLinux/Exemples/33
                                    fin
                                    Connection closed by foreign host.
                                    $
$
```

Nous voyons bien que le problème qui se pose est que la liaison par réseau entre `telnet` – qui peut gérer un terminal correctement – et notre serveur n'est pas suffisante pour autoriser le fonctionnement normal d'applications manipulant l'écran en mode brut.

Pour résoudre ce problème, nous devons faire appel à un pseudo-terminal.

Utilisation d'un pseudo-terminal

La notion de pseudo-terminal est souvent un peu confuse. Il s'agit en fait d'un périphérique virtuel géré par le noyau, offrant deux moitiés distinctes :

- Le pseudo-terminal *maître* est un périphérique géré comme un descripteur de fichier habituel, sur lequel on utilise `open()`, `read()`, `write()`, `close()`.
- Le pseudo-terminal *esclave* est vu par les processus exactement comme un terminal normal. Tout ce qu'on écrit sur le pseudo-terminal maître est lisible sur l'esclave et, inversement, ce qui est écrit sur le pseudo-terminal esclave est immédiatement accessible du côté maître. De plus, le noyau gère une discipline de ligne sur le pseudo-terminal esclave.

Nous allons donc pouvoir mettre en place le mécanisme décrit sur la figure 33.2 pour autoriser des connexions réseau complètes.

Pendant longtemps l'accès aux pseudo-terminaux se faisait en ouvrant directement des descripteurs particuliers : `/dev/ptyXN` pour les terminaux maîtres, et `/dev/ttyXN` pour les pseudo-terminaux esclaves, où `X` correspondait à une lettre dans l'ensemble « `pqrstuvwxyzabcde` » et `N` à un chiffre hexadécimal compris entre `0` et `F`.

À partir du noyau Linux 2.2, les pseudo-terminaux dits *devpts* ont fait leur apparition, conformément aux spécifications Unix 98. Les pseudo-terminaux sont maintenant situés dans `/dev/pts`, à condition qu'une pseudo-partition de type `devpts` soit montée sur ce répertoire.

Pour ouvrir un pseudo-terminal, on commence par invoquer la routine **`getpt()`**, une extension Gnu déclarée dans `<stdlib.h>` :

```
int getpt (void);
```

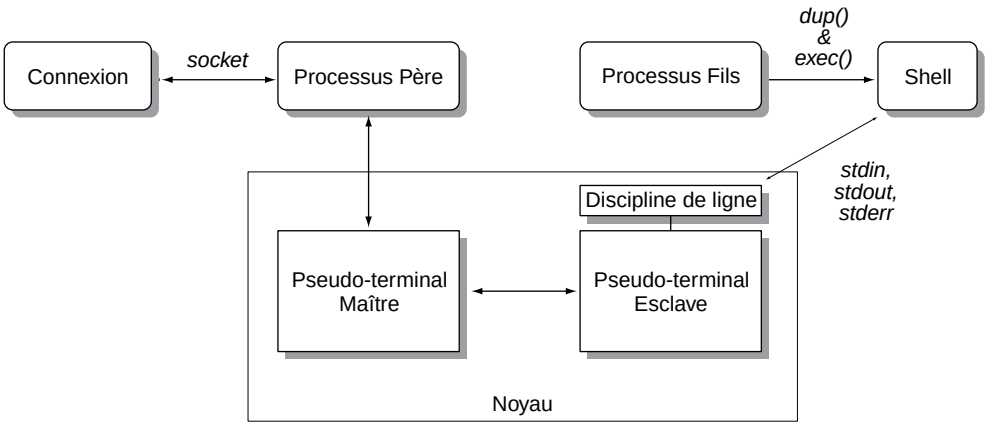


Figure 33.2
Utilisation d'un pseudo-terminal pour des connexions externes

Cette routine renvoie un descripteur de fichier correspondant au pseudo-terminal maître. Pour l'obtenir, elle demande au noyau d'ouvrir le fichier spécial `/dev/ptmx`. Voyant cela, le noyau lui attribue un descripteur d'un nouveau pseudo-terminal maître.

Une fois l'ouverture effectuée, il est nécessaire de modifier les droits d'accès au pseudo-terminal esclave – le seul qui sera vraiment visible dans l'arborescence du système de fichiers. Pour cela on appelle la fonction **`grantpt()`** en lui transmettant le descripteur du côté maître. Cette fonction est définie par SUSv3.

```
int grantpt (int descripteur);
```

Pour être sûr d'être portable, un programme doit ensuite appeler la routine **`unlockpt()`**, qui permet de déverrouiller le pseudo-terminal esclave associé au descripteur maître.

```
int unlockpt (int descripteur);
```

Finalement, pour pouvoir ouvrir le pseudo-terminal esclave, il faut obtenir son nom dans le système de fichiers. Ceci est assuré par la fonction **`ptsname()`**, qui renvoie un pointeur sur une chaîne de caractères en mémoire statique contenant le nom du pseudo-terminal esclave associé au descripteur maître passé en argument.

```
char * ptsname (int descripteur);
```

Il existe une extension Gnu réentrante nommée **`ptsname_r()`** :

```
int ptsname_r (int descripteur, char * buffer, size_t longueur);
```

Il ne faut pas être effrayé par la multitude de routines à invoquer successivement. L'utilisation des pseudo-terminaux répond à un schéma bien défini, qu'on réutilise directement dans chaque application.

Le principe de notre programme est le suivant :

- Le processus père ouvre un pseudo-terminal maître disponible. Ensuite il passe son pseudo-terminal en mode brut afin de ne pas interférer avec les échanges de données.

- Le fils ouvre le pseudo-terminal esclave associé au maître. Il en fait son terminal de contrôle et le duplique sur l'entrée et la sortie standard. Enfin il exécute un shell.
- Le processus père copie directement ce qu'il reçoit sur sa socket de communication vers le pseudo-terminal maître, et inversement.

> **Attention**
> Ce programme offre, sans vérification d'identité, un accès immédiat à la machine. Ne l'utilisez donc que sur une station se trouvant sur un réseau local sûr ou sur un compte utilisateur expérimental et sans privilèges.

Dans la description ci-dessus, le processus père considéré est déjà obtenu par un `fork()` après l'établissement de la connexion réseau. Nous avons donc deux niveaux de relations père-fils, mais nous ne nous intéressons ici qu'à la seconde, qui concerne les deux moitiés du pseudo-terminal.

exemple_pty.c :

```
    #define _GNU_SOURCE 500

    #include <fcntl.h>
    #include <limits.h>
    #include <signal.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <termios.h>
    #include <unistd.h>

    #include <arpa/inet.h>
    #include <netdb.h>
    #include <netinet/in.h>
    #include <sys/types.h>
    #include <sys/socket.h>

    void
copie_entrees_sorties (int fd, int sock)
{
    int    max;
    fd_set set;
    char   buffer[4096];
    int    nb_lus;

    max = sock < fd ? fd : sock;
    while (1) {
        FD_ZERO(& set);
        FD_SET(sock, & set);
        FD_SET(fd, & set);
        if (select(max + 1, & set, NULL, NULL, NULL) < 0)
            break;
        if (FD_ISSET(sock, &set)) {
```

```
            if ((nb_lus = read(sock, buffer, 4096)) >= 0)
                write(fd, buffer, nb_lus);
            else
                break;
        }
        if (FD_ISSET(fd, &set)) {
            if ((nb_lus = read(fd, buffer, 4096)) >= 0)
                write(sock, buffer, nb_lus);
            else
                break;
        }
    }
}

    void
traite_connexion (int sock)
{
    int    fd_maitre;
    int    fd_esclave;
    struct termios termios_stdin;
    struct termios termios_maitre;
    char * args[2] = { "/bin/sh", NULL };
    char * nom_esclave;

    if ((fd_maitre = getpt()) < 0) {
        perror("pas de Pseudo TTY Unix 98 disponibles \n");
        exit(EXIT_FAILURE);
    }
    grantpt(fd_maitre);
    unlockpt(fd_maitre);
    nom_esclave = ptsname(fd_maitre);
    tcgetattr(STDIN_FILENO, & termios_stdin);
    switch (fork()) {
        case -1 :
            perror("fork");
            exit(EXIT_FAILURE);
        case 0 : /* fils */
            close(fd_maitre);
            /* Détachement du terminal de contrôle précédent */
            setsid();
            /* Ouverture du pseudo-terminal esclave qui devient */
            /* alors le terminal de contrôle de ce processus.   */
            if ((fd_esclave = open(nom_esclave, O_RDWR)) < 0) {
                perror("open");
                exit(EXIT_FAILURE);
            }
            tcsetattr(fd_esclave, TCSANOW, & termios_stdin);
            dup2(fd_esclave, STDIN_FILENO);
            dup2(fd_esclave, STDOUT_FILENO);
            dup2(fd_esclave, STDERR_FILENO);
            execv(args[0], args);
```

```
            break;
        default :
            tcgetattr(fd_maitre, & termios_maitre);
            cfmakeraw(& termios_maitre);
            tcsetattr(fd_maitre, TCSANOW, & termios_maitre);
            copie_entrees_sorties(fd_maitre, sock);
            exit(EXIT_SUCCESS);
    }
}
```

La fonction `main()` reste inchangée par rapport au programme précédent. L'exécution du programme et la connexion réseau se déroulent déjà mieux qu'auparavant :

```
$ ./exemple_pty
Mon adresse : IP = 0.0.0.0, Port = 1231
                            $ telnet 192.1.1.51 1231
                            Trying 192.1.1.51...
                            Connected to 192.1.1.51.
                            Escape character is '^]'.
                            $ tty
                            tty
                            /dev/pts/7
                            $
                            $ su
                            su
                            Password: Visible !

                            su: incorrect password
                            $
                            $ exit
                            exit
                            exit
                            Connection closed by foreign host.
                            $
```

Le terminal de contrôle du shell lancé est bien dirigé vers un pseudo-terminal esclave, `/dev/pts/7` en l'occurrence. Nous remarquons toutefois un premier problème : les symboles d'invitation du shell (`$`) sont doublés. De plus, les commandes saisies sont répétées avant d'être exécutées. Un second problème se présente avec la saisie du mot de passe qui continue à être visible. Il y a donc un réglage de l'écho local qui n'est pas correct. Quant à l'utilisation d'un programme interactif comme `vi`, elle est très fortement perturbée (même si le processus a l'impression de travailler correctement sur un terminal).

Le véritable problème vient en fait de l'application `telnet`. Celle-ci est conçue pour dialoguer avec le démon `telnetd` et pas avec n'importe quel processus de connexion. Notre programme ne respecte pas le protocole TELNET (décrit entre autres dans la RFC 854).

Nous pouvons toutefois obtenir un résultat satisfaisant en écrivant notre propre processus client, qui copiera son entrée standard vers une socket réseau, et inversement copiera le contenu de la socket sur sa sortie standard. Il nous suffit de reprendre le programme `tcp_2_stdout.c` du chapitre précédent, dont nous modifions quelque peu la fonction `main()`.

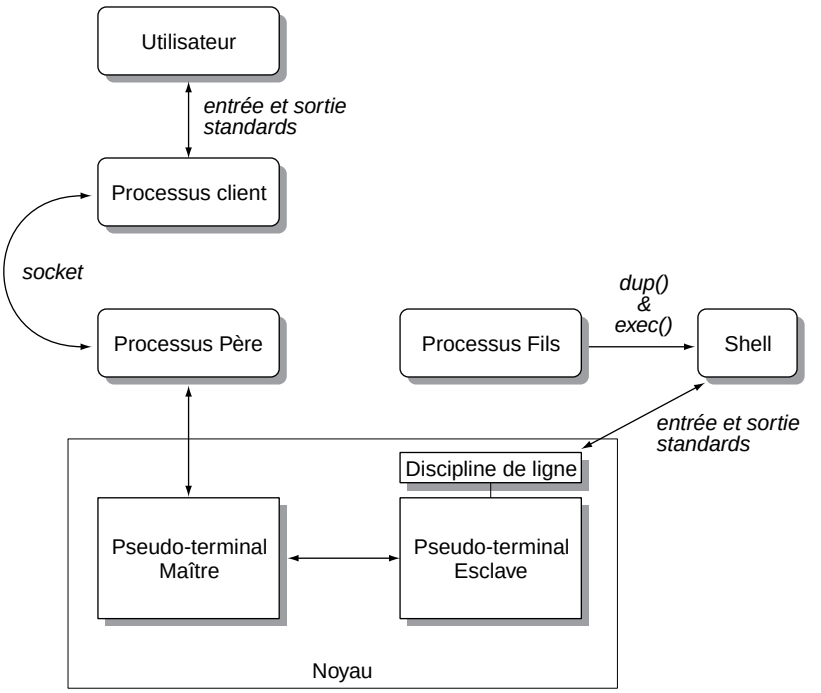
client_pty.c :

```
    int
main (int argc, char * argv[])
{
    int                 sock;
    struct sockaddr_in adresse;
    char                buffer[LG_BUFFER];
    int                 nb_lus;
    struct termios      termios_stdin, termios_raw;
    fd_set              set;

    if (lecture_arguments(argc, argv, & adresse, "tcp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (connect(sock, & adresse, sizeof(struct sockaddr_in)) < 0) {
        perror("connect");
        exit(EXIT_FAILURE);
    }
    tcgetattr(STDIN_FILENO, & termios_stdin);
    tcgetattr(STDIN_FILENO, & termios_raw);
    cfmakeraw(& termios_raw);
    tcsetattr(STDIN_FILENO, TCSANOW, & termios_raw);

    while (1) {
        FD_ZERO(& set);
        FD_SET(sock, & set);
        FD_SET(STDIN_FILENO, & set);
        if (select(sock + 1, & set, NULL, NULL, NULL) < 0)
            break;
        if (FD_ISSET(sock, & set)) {
            if ((nb_lus = read(sock, buffer, LG_BUFFER)) <= 0)
                break;
            write(STDOUT_FILENO, buffer, nb_lus);
        }
        if (FD_ISSET(STDIN_FILENO, & set)) {
            if ((nb_lus = read(STDIN_FILENO, buffer, LG_BUFFER)) <= 0)
                break;
            write(sock, buffer, nb_lus);
        }
    }
    tcsetattr(STDIN_FILENO, TCSANOW, & termios_stdin);
    return EXIT_SUCCESS;
}
```

Nous avons pris soin de basculer le terminal sur lequel s'exécute le client en mode non canonique, afin de laisser le shell se trouvant à l'autre extrémité de la chaîne responsable de la gestion de l'écran.

Nous avons donc réalisé un arrangement correspondant à la figure 33.3.

Figure 33.3
Organisation de notre connexion à distance



L'exécution cette fois-ci est parfaitement concluante, tous les programmes fonctionnent normalement, y compris les éditeurs plein écran comme `vi` ou des saisies de mot de passe caché.

```
$ ./exemple_pty
Mon adresse : IP = 0.0.0.0, Port = 1233
                                $ ./client_pty -a 192.1.1.51 -p 1233
                                $ tty
                                /dev/pts/7
                                $ su
                                Password:
                                # tty
                                /dev/pts/7
                                # exit
                                $ exit
                                exit
                                $
  (Contrôle-C)
$
```

Nous avons pu obtenir un système assez intéressant en utilisant des pseudo-terminaux pour offrir une possibilité de connexion distante. Nous allons examiner maintenant le dernier type principal d'utilisation des terminaux Unix : les liaisons série.

Configuration d'un port série RS-232

L'exploitation des ports série d'un PC sous Linux peut être envisagée pour de nombreuses raisons. Voici quelques exemples d'application que j'utilise fréquemment :

- Modem externe : ces modems ont en effet de nombreux avantages par rapport à leurs homologues internes, ne serait-ce que l'affichage visible de l'état des lignes de contrôle. On peut facilement les installer en rack dans des baies industrielles, et leur réinitialisation se fait sans arrêt de la machine.
- Appareil photographique numérique : l'application Gimp est très puissante en ce qui concerne le traitement des images, et de surcroît le rapatriement des photographies par câble série est nettement plus rapide avec des utilitaires sous Linux qu'avec les drivers d'origine livrés avec l'appareil.
- Imprimante : on trouve facilement des imprimantes d'occasion à ruban qui ont été remplacées par des imprimantes laser. Ces anciens modèles disposaient en général d'un système d'entraînement à picot permettant d'imprimer du listing au kilomètre. Cette possibilité est très précieuse pour le développeur, car la lecture d'un programme est plus cohérente sur ces feuilles en continu. Ces modèles d'imprimante existent souvent en version série, ce qui permet de conserver le port parallèle pour une imprimante bureautique plus classique.
- Câble de liaison : lors d'interventions sur des sites clients, il n'est pas toujours possible de connecter un ordinateur portable sur le réseau pour transférer des données, ceci pour des raisons de sécurité. La solution la plus simple pour transmettre des fichiers source est une liaison directe entre le port série du portable et celui de la station de destination. La mise en œuvre, nous le verrons, est facile et n'est généralement pas considérée comme une atteinte à la sécurité du système du client.
- Périphériques divers : l'interface RS-232 étant bien connue, assez performante et souple d'utilisation, il est fréquent de la rencontrer dans du matériel fabriqué spécifiquement pour un développement « maison ». J'ai utilisé cette interface pour communiquer avec des équipements allant du récepteur GPS à l'automate programmable, en passant par des centrales d'alarme ou des programmateurs de chauffage.

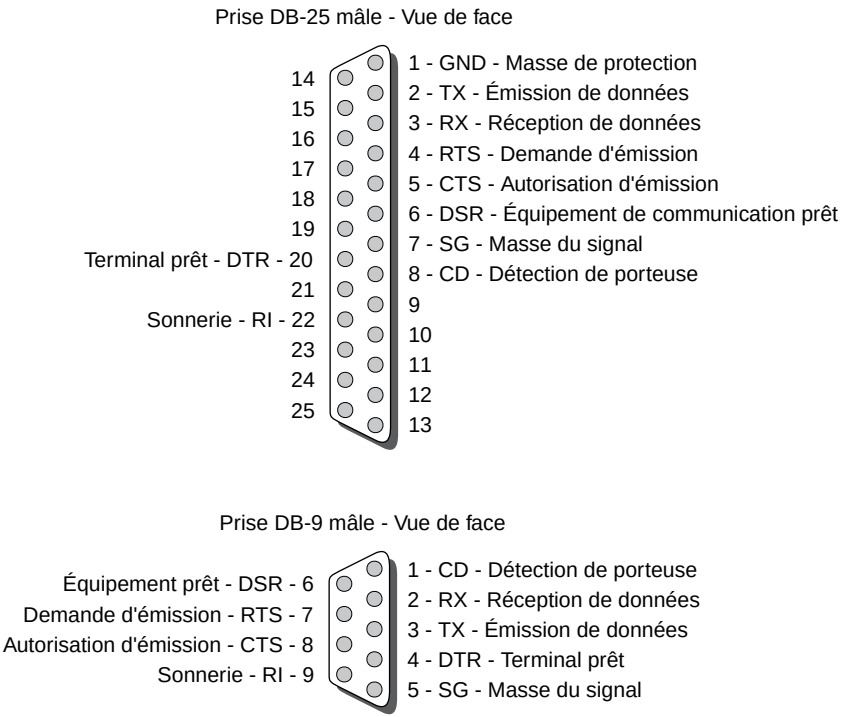
> Même si depuis la première édition de ce livre, l'utilisation des ports RS-232 a diminué au profit des ports USB, j'ai décidé de conserver ce paragraphe, car il existe encore de nombreux systèmes installés qui utilisent ce type de périphériques. Bien que les nouveaux ordinateurs ne soient pas toujours équipés de ports RS-232, des convertisseurs USB/RS-232 existent (et fonctionnent parfaitement sous Linux) qui nous permettent encore de réaliser ce genre de liaison.

Les ports série se présentent sur l'ordinateur sous forme de connecteurs mâles à 9 ou 25 broches, dont les noms sont indiqués sur la figure 33.4.

L'accès aux ports est possible en utilisant les fichiers spéciaux en mode caractère `/dev/ttyS0`, `/dev/ttyS1`…, ainsi que `/dev/cua0`, `/dev/cua1`… Les fichiers `/dev/ttyS0` et `/dev/cua0` correspondent au premier port série de l'unité centrale (noté *port A*, *COM1*…), `/dev/ttyS1` et `/dev/cua1` au second port, et ainsi de suite.

La différence entre les fichiers `/dev/ttySx` et `/dev/cuax` apparaît lors de l'ouverture du port. Si le port n'est pas configuré en mode *local* (option `CLOCAL` dans le membre `c_cflag` de la structure `termios`), l'ouverture d'un périphérique `/dev/ttySx` est bloquante en attendant que sa broche CD (détection de porteuse) passe à 1. Si le port est *local*, l'ouverture n'est pas bloquante. De même, l'ouverture de `/dev/cuax` n'est jamais bloquante.

Figure 33.4
Brochage des prises RS-232



L'emploi des périphériques `/dev/cuax` est à présent déconseillé, il vaut mieux procéder en ouvrant le port `/dev/ttySx` correspondant de manière non bloquante. En fait, `/dev/cuax` était réservé aux processus effectuant des connexions sortantes à l'aide d'un modem. L'ouverture ne devait pas être bloquante, même si aucune porteuse n'était détectée, car il fallait pouvoir demander au modem de réaliser la numérotation.

Nous respecterons les nouvelles consignes d'utilisation des ports série sous Linux en employant uniquement les `/dev/ttySx`.

Le premier problème est de trouver quel fichier spécial correspond à tel ou tel connecteur. Cela est évident lorsque les ports sont numérotés sur le panneau arrière du PC, mais lorsqu'une dizaine de machines sont installées en rack et reliées à des prolongateurs série de plusieurs mètres passant dans un faux plancher, l'étiquetage devient plus problématique. Pour cela, on peut toutefois utiliser les signaux de contrôle de la ligne. Lors de l'ouverture d'un port série, l'ordinateur élève les signaux RTS et DTR. Nous pouvons donc observer la modification électrique sur le connecteur au moyen d'un voltmètre ou d'un testeur à LED. Les tensions utilisées sur une prise RS-232 sont considérées comme des 0 logiques si elles sont inférieures à -3 V, et comme des 1 logiques si elles sont supérieures à +3 V, ceci par rapport à la borne SG de masse des signaux.

Notre premier programme va donc ouvrir le port indiqué en argument de manière non bloquante, puis le refermer en indiquant au fur et à mesure les tensions qu'on doit mesurer sur le connecteur.

exemple_serie.c :

```
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    #include <stdio.h>
    #include <unistd.h>

    int
main (int argc, char * argv[])
{
    int   fd;
    char  chaine[2];

    if (argc != 2) {
        fprintf(stderr, "%s < fichier spécial >\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Vérifiez la tension entre les broches \n"
                    "  7 (-) et 20 (+) pour un connecteur DB-25 \n"
                    "  5 (-) et  4 (+) pour un connecteur DB-9 \n \n ");
    fprintf(stdout, "La tension doit être inférieure à -3 V \n");
    fprintf(stdout, "Pressez Entrée pour continuer \n");
    fgets(chaine, 2, stdin);
    fd = open(argv[1], O_RDONLY | O_NONBLOCK);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "La tension doit être supérieure à +3 V \n");
    fprintf(stdout, "Pressez Entrée pour continuer \n");
    fgets(chaine, 2, stdin);
    fprintf(stdout, "La tension doit être à nouveau < -3 V \n");
    if (close(fd) < 0) {
        perror("close");
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS;
}
```

Pour mesurer la tension sur un connecteur série à l'aide d'un voltmètre, il est souvent plus facile d'y enficher un connecteur à souder de l'autre genre, sans câblage. Les bornes à souder se trouvant au dos de ce connecteur peuvent accueillir les pointes de mesure du voltmètre en évitant les dérapages constants.

Il faut noter que les fichiers spéciaux de périphériques comme `/dev/ttyS0` disposent d'autorisations d'accès souvent restrictives. Pour continuer nos expériences, il faut modifier les permissions pour donner l'accès à tous les utilisateurs (à éviter sur un système public), ou créer un groupe particulier ayant les droits de lecture et écriture et inscrire dans ce groupe les utilisateurs habilités à manipuler le port série.

À présent que nous avons trouvé le connecteur correspondant à notre fichier spécial, nous allons essayer de transférer des fichiers d'un ordinateur à l'autre. Pour cela il faut configurer les divers éléments de la liaison série :

- La parité est configurée par l'association des options `PARENB` et `PARODD` du membre `c_cflag` de la structure `termios`.
- Le nombre de bits de données est défini par les options `CS5`, `CS6`, `CS7` ou `CS8` du champ `c_cflag`. Avant de fixer une valeur, on efface tous les bits correspondant à ces options à l'aide du masque `CSIZE`.
- Le nombre de bits d'arrêt est fourni par l'option `CSTOPB` du membre `c_cflag`.
- La vitesse de transmission est en réalité contenue dans le champ `c_cflag`, mais on utilise des routines spécialisées pour la lire ou la configurer.

Les fonctions **`cfsetispeed()`** et **`cfsetospeed()`** permettent de configurer la vitesse d'entrée ou de sortie dans la structure `termios` passée en argument.

```
int cfsetispeed (struct termios * configuration, speed_t vitesse);
int cfsetospeed (struct termios * configuration, speed_t vitesse);
```

Le type **`speed_t`** représente la vitesse et peut prendre l'une des valeurs suivantes :

`B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, `B38400`, `B57600` ou `B115200`. Naturellement, chaque constante représente la vitesse correspondante mesurée en bits par seconde. La vitesse `B0` sert à forcer le raccrochage d'un modem.

Pour lire la vitesse configurée dans une structure `termios`, on peut employer **`cfgetispeed()`** ou **`cfgetospeed()`** :

```
speed_t cfgetispeed (struct termios * configuration);
speed_t cfgetospeed (struct termios * configuration);
```

Pour faire dialoguer deux ordinateurs, nous utiliserons le même principe que ce que nous avions élaboré avec les sockets UDP, en transférant sur une liaison série le contenu de l'entrée standard, et inversement depuis la liaison vers la sortie standard.

Le programme suivant va recopier son entrée standard vers un port série indiqué en argument. On commence par ouvrir le fichier spécial de manière non bloquante pour supprimer l'attribut *local* du port, puis on le referme. Lors de la seconde ouverture, le processus attendra que sa broche CD indique qu'une porteuse a été détectée.

stdin_2_serie.c :

```
    #include <fcntl.h>
    #include <stdio.h>
    #include <termios.h>
    #include <unistd.h>

    #define LG_BUFFER    1024

    void
setspeed (struct termios * config, speed_t vitesse)
{
    cfsetispeed (config, vitesse);
    cfsetospeed (config, vitesse);
}
```

```
    int
main (int argc, char * argv[])
{
    char *  nom_tty           = "/dev/ttyS0";
    int     vitesse           = 9600;
    int     type_parite       = 'n';
    int     nb_bits_donnees   = 8;
    int     nb_bits_arret     = 1;
    int     fd_tty            = -1;
    struct termios configuration;
    struct termios sauvegarde;

    char    buffer[LG_BUFFER];
    int     nb_lus;
    int     option;

    opterr = 0;
    while ((option = getopt(argc, argv, "hv:p:d:a:t:")) != -1) {
        switch (option) {
            case 'v' :
                if ((sscanf(optarg, "%d", & vitesse) != 1)
                 || (vitesse < 50) || (vitesse > 115200)) {
                    fprintf(stderr, "Vitesse %s invalide \n", optarg);
                    exit(EXIT_FAILURE);
                }
                break;
            case 'p' :
                type_parite = optarg[0];
                if ((type_parite != 'n') && (type_parite != 'p')
                 && (type_parite != 'i')) {
                    fprintf(stderr, "Parité %c invalide \n",
                                        type_parite);
                    exit(EXIT_FAILURE);
                }
                break;
            case 'd' :
                if ((sscanf(optarg, "%d", & nb_bits_donnees) != 1)
                 || (nb_bits_donnees < 5) || (nb_bits_donnees > 8)) {
                    fprintf(stderr, "Nb bits données %d invalide \n",
                                        nb_bits_donnees);
                    exit(EXIT_FAILURE);
                }
                break;
            case 'a' :
                if ((sscanf(optarg, "%d", & nb_bits_arret) != 1)
                 || (nb_bits_arret < 1) || (nb_bits_arret > 2)) {
                    fprintf(stderr, "Nb bits arrêt %d invalide \n",
                                    nb_bits_arret);
                    exit(EXIT_FAILURE);
                }
                break;
```

```
            case ‘t’ :
                nom_tty = optarg;
                break;
            case ‘h’ :
                fprintf(stderr, "Syntaxe %s [options]... \n", argv [0]);
                fprintf(stderr, "  Options : \n");
                fprintf(stderr, "     -v <vitesse en bits/seconde>  \n");
                fprintf(stderr, "     -p <parité> (n)ulle (p)aire "
                                                (i)mpaire \n");
                fprintf(stderr, "     -d <bits de données> (5 à 8) \n");
                fprintf(stderr, "     -a <bits d’arrêt> (1 ou 2) \n");
                fprintf(stderr, "     -t <nom du périphérique> \n");
                exit(EXIT_SUCCESS);
            default :
                fprintf(stderr, "Option -h pour avoir de l’aide \n");
                exit(EXIT_FAILURE);
        }
    }
    /* Ouverture non bloquante pour basculer en mode non local */
    fd_tty = open(nom_tty, O_RDWR | O_NONBLOCK);
    if (fd_tty < 0) {
        perror(nom_tty);
        exit(EXIT_FAILURE);
    }
    if (tcgetattr(fd_tty, & configuration) != 0) {
        perror("tcgetattr");
        exit(EXIT_FAILURE);
    }
    configuration.c_cflag &= ~ CLOCAL;
    tcsetattr(fd_tty, TCSANOW, & configuration);

    /* Maintenant ouverture bloquante en attendant CD */
    fd_tty = open(nom_tty, O_RDWR);
    if (fd_tty < 0) {
        perror(nom_tty);
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "Port série ouvert \n");
    tcgetattr(fd_tty, & configuration);
    memcpy(& sauvegarde, & configuration, sizeof(struct termios));
    cfmakeraw(& configuration);
    if (vitesse < 50)             setspeed(& configuration, B50);
    else if (vitesse < 75)        setspeed(& configuration, B75);
    else if (vitesse < 110)       setspeed(& configuration, B110);
    else if (vitesse < 134)       setspeed(& configuration, B134);
    else if (vitesse < 150)       setspeed(& configuration, B150);
    else if (vitesse < 200)       setspeed(& configuration, B200);
    else if (vitesse < 300)       setspeed(& configuration, B300);
    else if (vitesse < 600)       setspeed(& configuration, B600);
    else if (vitesse < 1200)      setspeed(& configuration, B1200);
    else if (vitesse < 1800)      setspeed(& configuration, B1800);
    else if (vitesse < 2400)      setspeed(& configuration, B2400);
```

```
else if (vitesse < 4800)      setspeed(& configuration, B4800);
else if (vitesse < 9600)      setspeed(& configuration, B9600);
else if (vitesse < 19200)     setspeed(& configuration, B19200);
else if (vitesse < 34000)     setspeed(& configuration, B38400);
else if (vitesse < 57600)     setspeed(& configuration, B57600);
else                          setspeed(& configuration, B115200);
switch (type_parite) {
    case 'n' :
        configuration.c_cflag &= ~ PARENB;
        break;
    case 'p' :
        configuration.c_cflag |=   PARENB;
        configuration.c_cflag &= ~ PARODD;
        break;
    case 'i':
        configuration.c_cflag |=   PARENB;
        configuration.c_cflag |=   PARODD;
        break;
}
configuration.c_cflag &= ~ CSIZE;
if (nb_bits_donnees == 5)         configuration.c_cflag |= CS5;
else if (nb_bits_donnees == 6)    configuration.c_cflag |= CS6;
else if (nb_bits_donnees == 7)    configuration.c_cflag |= CS7;
else if (nb_bits_donnees == 8)    configuration.c_cflag |= CS8;
if (nb_bits_arret == 1)           configuration.c_cflag &= ~ CSTOPB;
else                              configuration.c_cflag |=   CSTOPB;

configuration.c_cflag &= ~ CLOCAL;
configuration.c_cflag |=   HUPCL;

/* Contrôle de flux CTS/RTS spécifique Linux */
configuration.c_cflag |=   CRTSCTS;

if (tcsetattr(fd_tty, TCSANOW, & configuration) < 0) {
    perror("tcsetattr");
    exit(EXIT_FAILURE);
}
fprintf(stderr, "Port série configuré \n");

fprintf(stderr, "Début de l'envoi des données \n");
while (1) {
    nb_lus = read(STDIN_FILENO, buffer, LG_BUFFER);
    if (nb_lus < 0)  {
        perror("read");
        exit(EXIT_FAILURE);
    }
    if (nb_lus == 0)
        break;
    write(fd_tty, buffer, nb_lus);
}
fprintf(stderr, "Fin de l'envoi des données \n");
sleep(2);
```

```
    close(fd_tty);
    /* restauration de la configuration originale */
    fd_tty = open(nom_tty, O_RDWR | O_NONBLOCK);
    sauvegarde.c_cflag |= CLOCAL;
    tcsetattr(fd_tty, TCSANOW, & sauvegarde);
    close(fd_tty);
    return EXIT_SUCCESS;
}
```

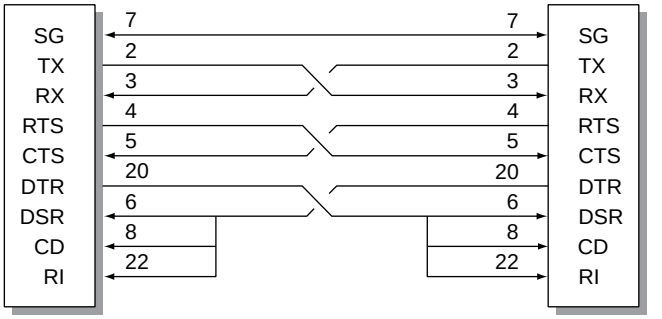
Dans le programme `serie_2_stdout.c`, seule la partie centrale de la routine `main()` a été modifiée :

serie_2_stdout.c :

```
    fprintf(stderr, "Début de la réception des données \n");
    while (1) {
        nb_lus = read(fd_tty, buffer, LG_BUFFER);
        if (nb_lus < 0)  {
            perror("read");
            exit(EXIT_FAILURE);
        }
        if (nb_lus == 0)
            break;
        write(STDOUT_FILENO, buffer, nb_lus);
    }
    fprintf(stderr, "Fin de la réception des données \n");
    close(fd_tty);
```

Pour se servir de ces utilitaires, il faut disposer d'un câble de liaison, dit *Null-Modem*, qui croise les lignes de données et de contrôle, et simule la détection de porteuse lorsque l'autre ordinateur est prêt. La figure 33.5 montre un exemple de ce câble, qu'on trouve parfois sous le nom de connexion DTE-DTE complète.

Figure 33.5
Câble Null-Modem complet avec brochage DB25



Voici un exemple d'exécution sur deux ordinateurs reliés par leurs premiers ports série :

```
$ ./stdin_2_serie -t /dev/ttyS0 < stdin_2_serie.c
                $ ./serie_2_stdout -t /dev/ttyS0 > stdin_2_serie.c
                Port série ouvert
```

```
Port série ouvert
                Port série configuré
Port série configuré
                Début de la réception des données
Début de l'envoi des données
Fin de l'envoi des données
                Fin de la réception des données
                $ ls stdin_2_serie.c
                stdin_2_serie.c
                $ cksum stdin_2_serie.c
                2971580910 5399 stdin_2_serie.c
                $
$ ls stdin_2_serie.c
stdin_2_serie.c
$ cksum stdin_2_serie.c
2971580910 5399 stdin_2_serie.c
$
```

L'utilitaire `cksum` invoqué ici calcule une somme de contrôle représentant une sorte de signature du fichier indiqué. On vérifie ainsi aisément que les données ont bien été transmises d'un ordinateur à l'autre. Nos deux programmes ne représentent que des exemples très simplifiés pour le transfert de fichiers. Si on voulait en faire de véritables outils sérieux, il faudrait vérifier les conditions d'erreur à la réception et utiliser un protocole permettant de demander à l'émetteur de renvoyer à nouveau un paquet de données erroné.

Conclusion

Nous avons examiné dans ce dernier chapitre les aspects les plus utiles de la gestion des terminaux. En ce qui concerne l'utilisation de fonctionnalités étendues pour le terminal (édition plein écran, etc.), on emploiera de préférence la bibliothèque *ncurses*, qui offre de nombreuses possibilités et sait gérer l'essentiel des terminaux courants.

Les lecteurs désireux d'approfondir le sujet sur les pseudo-terminaux pourront se tourner vers [STEVENS 1993] *Advanced Programming in the UNIX Environment*.

Pour les liaisons série, on trouvera des renseignements dans de nombreux ouvrages, en particulier dans [NELSON 1994] *Communications série, guide du développeur C++*, même s'il est plutôt orienté vers le monde Dos. La configuration de ces liaisons pour des terminaux Unix est abordée dans [FRISCH 2003] *Les bases de l'administration système*.

Les liaisons RS-232 sont aussi décrites en détail dans les documents Linux *Serial-HOWTO*, *Serial-Programming-HOWTO* et *Modems-HOWTO*.

Annexe 1

Tables Ascii et ISO 8859-1/15

Première moitié : Ascii								
0x00	\0	Ctrl-A	Ctrl-B	Ctrl-C	Ctrl-D	Ctrl-E	Ctrl-F	\a
0x10	Ctrl-P	Ctrl-Q	Ctrl-R	Ctrl-S	Ctrl-T	Ctrl-U	Ctrl-V	Ctrl-W
0x18	Ctrl-X	Ctrl-Y	Ctrl-Z	(Esc)				
0x20	Espace	!	"	#	$	%	&	'
0x28	(	)	*	+	,	-	.	/
0x30	0	1	2	3	4	5	6	7
0x38	8	9	:	;	<	=	>	?
0x40	@	A	B	C	D	E	F	G
0x48	H	I	J	K	L	M	N	O
0x50	P	Q	R	S	T	U	V	W
0x58	X	Y	Z	[	\	]	^	_
0x60	`	a	b	c	d	e	f	g
0x68	h	i	j	k	l	m	n	o
0x70	p	q	r	s	t	u	v	w
0x78	x	y	z	{	|	}	~	(DEL)

Seconde moitié : ISO 8859-1								
0x80								
0x88								
0x90								
0x98								
0xA0		¡	¢	£	¤	¥	¦	§
0xA8	¨	©	ª	«	¬	–	®	¯
0xB0	°	±	²	³	´	µ	¶	·
0xB8	¸	¹	º	»	¼	½	¾	¿
0xC0	À	Á	Â	Ã	Ä	Å	Æ	Ç
0xC8	È	É	Ê	Ë	Ì	Í	Î	Ï
0xD0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×
0xD8	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
0xE0	à	á	â	ã	ä	å	æ	ç
0xE8	è	é	ê	ë	ì	í	î	ï
0xF0	ð	ñ	ò	ó	ô	õ	ö	÷
0xF8	ø	ù	ú	û	ü	ý	þ	ÿ

Le codage ISO 8859-15 est un dérivé de l'ISO 8859-1, auquel on a ajouté le symbole « Euro ». De plus quelques caractères ont été modifiés dans les lignes suivantes :

0xA0		¡	¢	£	€	¥	Š	§
0xA8	š	©	ª	«	¬	–	®	¯
0xB0	°	±	²	³	Ž	µ	¶	·
0xB8	ž	¹	º	»	Œ	œ	Ÿ	¿

Annexe 2

Fonctions et appels-système étudiés

On trouvera ci-dessous une liste alphabétique des fonctions et des appels-système qui ont été présentés au cours de cet ouvrage. On y trouvera aussi le numéro du chapitre dans lequel on a étudié la fonction, avec une indication précisant s'il s'agit d'une fonction de la bibliothèque C ou d'un appel-système. On y signale également si la routine est mentionnée dans les normes courantes (Iso C9X, SUSv3). Les extensions spécifiques Gnu sont relevées, ainsi que les fonctions désormais considérées comme obsolètes.

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
abort	5	•		•	•		
abs	24	•		•	•		
accept	32		•		•		
access	21		•		•		
acos	24	•		•	•		
acosh	24	•		•	•		
adjtime	25	•				•	
adjtimex	25		•				
addmntent	26	•				•	
aio_cancel	30	•			•		
aio_error	30	•			•		
aio_fsync	30	•			•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
aio_read	30	•			•		
aio_return	30	•			•		
aio_suspend	30	•			•		
aio_write	30	•			•		
alarm	7		•		•		
alloca	13	•				•	
alphasort	20	•				•	
asctime	25	•			•		
asctime_r	25	•		•	•		
asin	24	•		•	•		
asinh	24	•			•		
assert	5	•		•	•		
atan	24	•		•	•		
atan2	24	•		•	•		
atanh	24	•			•		
atexit	5	•		•	•		
atof	23	•		•	•		
atoi	23	•		•	•		
atol	23	•		•	•		
atoll	23	•				•	
basename	15	•			•		
bcmp	15	•			•		•
bcopy	15	•			•		•
bind	32		•		•		
bindtextdomain	27	•				•	
brk	13		•				
bsearch	17	•		•	•		
btowc	23	•			•		
bzero	15	•			•		•
cabs	24	•		•	•		
cacos	24	•		•	•		
cacosh	24	•		•	•		
calloc	13	•		•	•		
capget	2		•				
capset	2		•				
carg	24	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
casin	24	•		•	•		
casinh	24	•		•	•		
catan	24	•		•	•		
catanh	24	•		•	•		
catclose	27	•			•		
catgets	27	•			•		
catopen	27	•			•		
cbc_crypt	16	•				•	
cbrt	24	•			•		
ccos	24	•		•	•		
ccosh	24	•		•	•		
ceil	24	•		•	•		
cexp	24	•		•	•		
cfgetispeed	33	•			•		
cfgetospeed	33	•			•		
cfmakeraw	33	•					
cfsetispeed	33	•			•		
cfsetospeed	33	•			•		
chdir	20		•		•		
chmod	21		•		•		
chown	21		•		•		
chroot	20		•		•		
cimag	24	•		•	•		
clearenv	3	•				•	
clearerr	18	•		•	•		
clock	9	•		•	•		
clog	24	•		•	•		
__clone	12		•				
close	19		•		•		
closedir	20	•			•		
closelog	26	•			•		
conj	24	•		•	•		
connect	32		•		•		
copysign	24	•				•	
cos	24	•		•	•		
cosh	24	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
cpow	24	•		•	•		
creal	24	•		•	•		
creat	19		•		•		
crypt	16	•			•		
crypt_r	16	•				•	
csin	24	•		•	•		
csinh	24	•		•	•		
csqrt	24	•		•	•		
ctan	24	•		•	•		
ctanh	24	•		•	•		
ctermid	33	•			•		
ctime	25	•		•	•		
ctime_r	25	•			•		
cuserid	26	•					•
dbmclose	22	•			•		•
dbminit	22	•			•		•
dbm_clearerr	22	•			•		
dbm_close	22	•			•		
dbm_delete	22	•			•		
dbm_dirfno	22	•			•		
dbm_error	22	•			•		
dbm_fetch	22	•			•		
dbm_firstkey	22	•			•		
dbm_nextkey	22	•			•		
dbm_open	22	•			•		
dbm_pagfno	22	•			•		
dbm_rdonly	22	•			•		
dbm_store	22	•			•		
dbopen	22	•					
delete	22	•					•
des_setparity	16	•				•	
difftime	25	•		•	•		
div	24	•		•	•		
drand48	24	•			•		
drand48_r	24	•				•	
drem	24	•				•	

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
dup	19		•		•		
dup2	19		•		•		
ebc_crypt	16	•				•	
ecvt	23	•			•		
ecvt_r	23	•				•	
encrypt	16	•			•		
encrypt_r	16	•				•	
endfsent	26	•				•	
endgrent	26	•			•		
endhostent	31	•			•		
endmntent	26	•				•	
endnetent	31	•			•		
endprotoent	31	•			•		
endpwent	26	•			•		
endservent	31	•			•		
endusershell	26	•					
endutent	26	•				•	
endutxent	26	•			•		
erand48	24	•			•		
erand48_r	24	•				•	
erf	24	•			•		
erfc	24	•			•		
errno	5	•		•	•		
execl	4	•			•		
execle	4	•			•		
execlp	4	•			•		
execv	4	•			•		
execve	4		•		•		
execvp	4	•			•		
_exit	5		•		•		
exit	5	•		•	•		
exp	24	•		•	•		
exp2	24			•	•		
exp10	24			•		•	
expm1	24	•			•		
fabs	24	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
fchdir	20		•		•		
fchmod	21		•		•		
fchown	21		•		•		
fclose	18	•		•	•		
fcloseall	18	•				•	
fcntl	19		•		•		
fcvt	23	•			•		
fcvt_r	23	•				•	
fdatasync	30		•		•		
fdopen	18	•			•		
feof	18	•		•	•		
ferror	18	•		•	•		
fetch	22	•			•		•
fflush	18	•		•	•		
fgetc	10			•	•		
fgetgrent	26	•					
fgetgrent_r	26	•				•	
fgetpos	18	•		•	•		
fgetpwent	26	•					
fgetpwent_r	26	•					
fgets	10			•	•		
fgetwc	23	•		•	•		
fgetws	23	•		•	•		
fileno	18	•			•		
finite	24	•				•	
first_key	22	•			•		•
flock	19		•				•
floor	24	•		•	•		
fmod	24	•		•	•		
fnmatch	20	•			•		
fopen	18	•		•	•		
fork	2		•		•		
fprintf	10	•		•	•		
fputc	10	•		•	•		
fputs	10	•		•	•		
fputwc	23	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
fputws	23	•		•	•		
fread	18	•		•	•		
free	13	•		•	•		
freopen	18	•		•	•		
frexp	24	•		•	•		
fscanf	10	•		•	•		
fseek	18	•		•	•		
fseeko	18	•			•		
fsetpos	18	•		•	•		
fstat	21		•	•	•		
fstatfs	26		•				
fsync	19		•		•		
ftell	18	•		•	•		
ftello	18	•			•		
ftime	25	•			•		•
ftok	29	•			•		
ftruncate	21		•		•		
ftw	20	•			•		
fwprintf	23	•		•	•		
fwscanf	23	•		•	•		
fwrite	18	•		•	•		
gcvt	23	•			•		
gdbm_close	22	•				•	
gdbm_delete	22	•				•	
gdbm_exist	22	•				•	
gdbm_fdesc	22	•				•	
gdbm_fetch	22	•				•	
gdbm_firstkey	22	•				•	
gdbm_nextkey	22	•				•	
gdbm_open	22	•				•	
gdbm_reorganize	22	•				•	
gdbm_setopt	22	•				•	
gdbm_store	22	•				•	
gdbm_strerror	22	•				•	
gdbm_sync	22	•				•	
getc	10	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
getchar	10	•		•	•		
getcwd	20	•			•		
getdate	25	•			•		
getdate_r	25	•				•	
getdomainname	26		•				
getegid	2		•		•		
getenv	3		•	•	•		
geteuid	2		•		•		
getfsent	26	•				•	
getfsfile	26	•				•	
getfsspec	26	•				•	
getgid	2	•			•		
getgrent	26	•			•		
getgrent_r	26	•				•	
getgrgid	26	•			•		
getgrgid_r	26	•				•	
getgrnam	26	•			•		
getgrnam_r	26	•			•		
getgroups	2		•		•		
gethostbyaddr	31	•			•		
gethostbyaddr_r	31	•				•	
gethostbyname	31	•			•		
gethostbyname_r	31	•				•	
gethostbyname2	31	•				•	
gethostbyname2_r	31	•				•	
gethostent	31	•			•		
gethostid	26		•		•		
gethostname	26		•		•		
getitimer	9		•		•		
getline	10	•				•	
getlogin	26	•			•		
getlogin_r	26	•			•		
getmntent	26	•				•	
getmntent_t	26	•				•	
getnetbyaddr	31	•			•		
getnetbyname	31	•			•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
getnetent	31	•			•		
getopt	3	•			•		
getopt_long	3	•				•	
getopt_long_only	3	•				•	
getpass	33	•					
getpeername	32		•		•		
getpgid	2		•		•	•	
getpgrp	2		•		•		
getpid	2		•		•		
getppid	2		•		•		
getpriority	11		•		•		
getprotobyname	31	•			•		
getprotobyname_r	31	•				•	
getprotobynumber	31	•			•		
getprotobynumber_r	31	•				•	
getprotoent	31	•			•		
getprotoent_r	31	•				•	
getpt	33	•				•	
getpwent	26	•			•		
getpwent_r	26	•				•	
getpwnam	26	•			•		
getpwnam_r	26	•			•		
getpwuid	26	•			•		
getpwuid_r	26	•			•		
getresgid	2		•				
getresuid	2		•				
getrlimit	9		•		•		
getrusage	9		•		•		
gets	10			•	•		•
getservbyname	31	•			•		
getservbyname_r	31	•				•	
getservbyport	31	•			•		
getservbyport_r	31	•				•	
getservent	31	•			•		
getservent_r	31	•				•	
getsid	2		•		•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
getsockname	32		•		•		
getsockopt	32		•		•		
getsubopt	3	•			•		
gettext	27	•				•	
getuid	2		•		•		
getusershell	26	•					
getutent	26	•				•	
getutent_r	26	•				•	
getutid	26	•				•	
getutid_r	26	•				•	
getutline	26	•				•	
getutline_r	26	•				•	
getutxent	26	•			•		
getutxid	26	•			•		
getutxline	26	•			•		
getw	18	•			•		•
getwc	23	•		•	•		
getwchar	23	•		•	•		
getwd	20	•			•		
glob	20	•			•		
globfree	20	•			•		
gmtime	25	•		•	•		
gmtime_r	25	•			•		
grantpt	33	•			•		
hasmntopt	26	•				•	
hcreate	17	•			•		
hcreate_r	17	•				•	
hdestroy	17	•			•		
hdestroy_r	17	•				•	
herror	31	•					•
hstrerror	31	•					•
hsearch	17	•			•		
hsearch_r	17	•				•	
htonl	31	•			•		
htons	31	•			•		
hypot	24	•			•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
index	15	•			•		•
inet_addr	31	•			•		
inet_aton	31	•					
inet_lnaof	31	•			•		
inet_netof	31	•			•		
inet_ntoa	31	•			•		
inet_ntop	31	•			•		
inet_pton	31	•			•		
infnan	24	•					
initgroups	26	•					
initstate	24	•			•		•
ioctl	33		•		•		
isalnum	23	•		•	•		
isalpha	23	•		•	•		
isascii	23	•			•		
isatty	33	•			•		
isblank	23	•		•	•		
iscntrl	23	•		•	•		
isdigit	23	•		•	•		
isgraph	23	•		•	•		
isinf	24	•			•		
islower	23	•		•	•		
isnan	24	•			•		
isprint	23	•		•	•		
ispunct	23	•		•	•		
isspace	23	•		•	•		
isupper	23	•		•	•		
iswalnum	23	•		•	•		
iswalpha	23	•		•	•		
iswblank	23	•		•	•		
iswcntrl	23	•		•	•		
iswdigit	23	•		•	•		
iswgraph	23	•		•	•		
iswlower	23	•		•	•		
iswprint	23	•		•	•		
iswpunct	23	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
iswspace	23	•		•	•		
iswupper	23	•		•	•		
iswxdigit	23	•		•	•		
isxdigit	23	•		•	•		
j0	24	•			•		
j1	24	•			•		
jn	24	•			•		
jrand48	24	•			•		
jrand48_r	24	•				•	
kill	6		•		•		
killpg	6		•		•		
labs	24	•		•	•		
lchown	21		•		•		
lcong48	24				•		
lcong48_r	24	•				•	
ldexp	24	•		•	•		
ldiv	24	•		•	•		
lfind	17	•			•		
lgamma	24	•			•		
link	21		•		•		
lio_listio	30	•			•		
listen	32		•		•		
localeconv	27	•		•	•		
localtime	25	•		•	•		
localtime_r	25	•			•		
log	24	•		•	•		
log2	24			•	•		
log10	24	•		•	•		
log1p	24	•			•		
longjmp	7	•		•	•		•
lrand48	24	•			•		
lrand48_r	24	•				•	
lsearch	17	•			•		
lseek	19		•		•		
lstat	21		•		•		
malloc	13	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
mallopt	13	•				•	
mblen	23	•			•		
mbrlen	23	•			•		
mbrtowc	23	•			•		
mbsnrtowcs	23	•				•	
mbsrtowcs	23	•			•		
mbstowcs	23	•			•		
mbtowc	23	•			•		
mcheck	13	•				•	
memccpy	15	•			•		
memchr	15	•		•	•		
memcmp	15	•		•	•		
memcpy	15	•		•	•		
memfrob	15	•				•	
memmem	15	•				•	
memmove	15	•		•	•		
mempcpy	15	•				•	
memset	15	•		•	•		
mkdir	20		•		•		
mkfifo	28	•			•		
mknod	21		•		•		
mkstemp	20	•			•		
mktemp	20	•			•		
mktime	25	•		•	•		
mlock	14		•		•		
mlockall	14		•		•		
mmap	14		•		•		
modf	24	•		•	•		
mprobe	13	•				•	
mprotect	14		•		•		
mrand48	24	•			•		
mrand48_r	24	•				•	
mremap	14		•				
msgctl	29		•		•		
msgget	29		•		•		
msgrcv	29		•		•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
msgsnd	29		•		•		
msync	14		•		•		
mtrace	13	•				•	
munlock	14		•		•		
munlockall	14		•		•		
munmap	14		•		•		
muntrace	13	•				•	
nanosleep	9		•		•		
next_key	22	•			•		•
nftw	20	•			•		
nice	11		•		•		
nl_langinfo	27	•			•		
nrand48	24	•			•		
nrand48_r	24	•				•	
ntohl	31	•			•		
ntohs	31	•			•		
on_exit	5	•					
open	19		•		•		
opendir	20	•			•		
openlog	26	•			•		
pause	7		•		•		
pclose	4				•		
personality	30		•				
perror	5			•	•		
pipe	28		•		•		
poll	30		•		•		
popen	4				•		
pow	24	•		•	•		
pread	19		•		•		
printf	10			•	•		
pselect	26		•		•		
psignal	6	•					
pthread_atfork	12	•			•		
pthread_attr_destroy	12	•			•		
pthread_attr_getdetachstate	12	•			•		
pthread_attr_getguardsize	12	•			•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
pthread_attr_getinheritsched	12	•			•		
pthread_attr_getshedparam	12	•			•		
pthread_attr_getschedpolicy	12	•			•		
pthread_attr_getscope	12	•			•		
pthread_attr_getstackaddr	12	•			•		
pthread_attr_getstacksize	12	•			•		
pthread_attr_init	12	•			•		
pthread_attr_setdetachstate	12	•			•		
pthread_attr_setguardsize	12	•			•		
pthread_attr_setinheritsched	12	•			•		
pthread_attr_setschedparam	12	•			•		
pthread_attr_setschedpolicy	12	•			•		
pthread_attr_setscope	12	•			•		
pthread_attr_setstackaddr	12	•			•		
pthread_attr_setstacksize	12	•			•		
pthread_cancel	12	•			•		
pthread_cleanup_pop	12	•			•		
pthread_cleanup_push	12	•			•		
pthread_cond_broadcast	12	•			•		
pthread_cond_destroy	12	•			•		
pthread_cond_init	12	•			•		
pthread_cond_signal	12	•			•		
pthread_cond_timedwait	12	•			•		
pthread_cond_wait	12	•			•		
pthread_condattr_destroy	12	•			•		
pthread_condattr_init	12	•			•		
pthread_create	12	•			•		
pthread_detach	12	•			•		
pthread_equal	12	•			•		
pthread_exit	12	•			•		
pthread_getschedparam	12	•			•		
pthread_getspecific	12	•			•		
pthread_join	12	•			•		
pthread_key_create	12	•			•		
pthread_key_delete	12	•			•		
pthread_kill	12	•			•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
pthread_mutex_destroy	12	•			•		
pthread_mutex_init	12	•			•		
pthread_mutex_lock	12	•			•		
pthread_mutex_trylock	12	•			•		
pthread_mutex_unlock	12	•			•		
pthread_mutexattr_destroy	12	•			•		
pthread_mutexattr_gettype	12	•			•		
pthread_mutexattr_init	12	•			•		
pthread_mutexattr_settype	12	•			•		
pthread_once	12	•			•		
pthread_self	12	•			•		
pthread_setcancelstate	12	•			•		
pthread_setcanceltype	12	•			•		
pthread_setschedparam	12	•			•		
pthread_setspecific	12	•			•		
pthread_sigmask	12	•			•		
pthread_testcancel	12	•			•		
ptsname	33	•			•		
ptsname_r	33	•				•	
putc	10	•		•	•		
putchar	10	•		•	•		
putenv	3	•			•		
putpwent	26	•					
puts	10	•		•	•		
pututline	26	•				•	
pututxline	26	•			•		
putw	18	•			•		•
putwc	23	•		•	•		
putwchar	23	•		•	•		
pwrite	19		•		•		
qecvt	23	•				•	
qecvt_r	23	•				•	
qfcvt	23	•				•	
qfcvt_r	23	•				•	
qsort	17	•		•	•		
raise	6	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
rand	24	•		•	•		
rand_r	24	•				•	
random	24	•			•		•
rawmemchr	15	•				•	
read	19		•		•		
readdir	20		•		•		
readlink	21		•		•		
readv	19		•		•		
realloc	13	•		•	•		
realpath	20	•			•		
re_comp	16	•			•		•
recv	32		•		•		
recvfrom	32		•		•		
recvmsg	32		•		•		
re_exec	16	•			•		•
regcomp	16	•			•		
regerror	16	•			•		
regexec	16	•			•		
regfree	16	•			•		
remove	20	•		•	•		
rename	20		•	•	•		
rewind	18	•		•	•		
rewinddir	20	•			•		
rindex	15	•			•		•
rint	24	•			•		
rmdir	20		•		•		
sbrk	13	•			•		
scandir	20	•			•		
scanf	10	•		•	•		
sched_getparam	11		•		•		
sched_get_priority_max	11		•		•		
sched_get_priority_min	11		•		•		
sched_getscheduler	11		•		•		
sched_rr_get_interval	11		•		•		
sched_setparam	11		•		•		
sched_setscheduler	11		•		•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
sched_yield	11		•		•		
seed48	24	•			•		
seed48_r	24	•				•	
seekdir	20	•			•		
select	26		•		•		
sem_close	12	•			•		
sem_destroy	12	•			•		
sem_getvalue	12	•			•		
sem_init	12	•			•		
sem_open	12	•			•		
sem_post	12	•			•		
sem_trywait	12	•			•		
sem_wait	12	•			•		
semctl	29		•		•		
semget	29		•		•		
semop	29		•		•		
send	32		•		•		
sendmsg	32		•		•		
sendto	32		•		•		
setbuf	18	•		•	•		
setbuffer	18	•					
setdomainname	26		•				
setegid	2		•				
setenv	3	•					
seteuid	2		•				
setfsent	26	•				•	
setgid	2		•		•		
setgrent	26	•			•		
setgroups	2		•				
sethostent	31	•			•		
sethostid	26		•				
sethostname	26		•				
setitimer	9		•		•		
setjmp	7	•		•	•		•
setkey	16	•				•	
setkey_r	16	•				•	

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
setlinebuf	18	•					
setlocale	27	•		•	•		
setmntent	26	•				•	
setnetent	31	•			•		
setpgid	2		•		•		
setpgrp	2		•		•		
setpriority	11		•		•		
setprotoent	31	•			•		
setpwent	26	•			•		
setregid	2		•		•		
setresgid	2		•				
setresuid	2		•				
setreuid	2		•		•		
setrlimit	9		•		•		
setservent	31	•			•		
setsid	2		•		•		
setsockopt	32		•		•		
setstate	24	•			•		•
settimeofday	25		•				
setuid	2		•		•		
setusershell	26	•					
setutent	26	•				•	
setutxent	26	•			•		
setvbuf	18			•	•		
shmat	29		•		•		
shmctl	29		•		•		
shmdt	29		•		•		
shmget	29		•		•		
shutdown	32		•		•		
sigaction	7		•		•		
sigaddset	7	•			•		
sigaltstack	7		•		•		
sigandset	7	•				•	
sigblock	7		•				•
sigdelset	7	•			•		
sigemptyset	7	•			•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
sigfillset	7	•			•		
siggetmask	7		•				•
siginterrupt	6	•			•	•	
sigisemptyset	7	•				•	
sigismember	7	•			•		
siglongjmp	7	•			•		
sigmask	7		•				•
signal	6		•	•	•		
sigorset	7	•				•	
sigpause	7		•		•		•
sigpending	7		•		•		
sigprocmask	7		•		•		
sigqueue	8		•		•		
sigsetjmp	7	•			•		
sigsetmask	7		•				•
sigsuspend	7		•		•		
sigtimedwait	8		•		•		
sigvec	6		•				•
sigwait	12	•			•		
sigwaitinfo	8		•		•		
sin	24	•		•	•		
sincos	24	•				•	
sinh	24	•		•	•		
sleep	9	•			•		
snprintf	10	•			•		
socket	32		•		•		
socketpair	32		•		•		
sprintf	10	•		•	•		
sqrt	24	•		•	•		
srand	24	•		•	•		
srand48	24	•			•		
srand48_r	24	•				•	
srandom	24	•			•		•
sscanf	10	•		•	•		
stat	21		•		•		
statfs	26		•				

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
stderr	10	•		•	•		
stdin	10	•		•	•		
stdout	10	•		•	•		
stime	25		•				
store	22	•			•		•
stpcpy	15	•				•	
stpncpy	15	•				•	
strcasecmp	15	•			•		
strcasestr	15	•				•	
strcat	15	•		•	•		
strchr	15	•		•	•		
strcmp	15	•		•	•		
strcoll	15	•		•	•		
strcpy	15	•		•	•		
strcspn	15	•		•	•		
strdup	15	•			•		
strdupa	15	•				•	
strerror	5	•		•	•		
strerror_r	5	•				•	
strfmon	27	•			•		
strfry	16	•				•	
strftime	25	•		•	•		
strlen	15	•		•	•		
strncasecmp	15	•			•		
strncat	15	•		•	•		
strncmp	15	•		•	•		
strncpy	15	•		•	•		
strnlen	15	•				•	
strndup	15	•				•	
strndupa	15	•				•	
strpbrk	15	•		•	•		
strptime	25	•			•		
strrchr	15	•		•	•		
strsep	15	•					
strsignal	6	•				•	
strspn	15	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
strstr	15	•		•	•		
strtod	23	•		•	•		
strtof	23	•		•		•	
strtok	15	•		•	•		
strtok_r	15	•			•	•	
strtol	23	•		•	•		
strtold	23	•		•		•	
strtoll	23	•		•		•	
strtoul	23	•		•	•		
strtoull	23	•		•		•	
strxfrm	15	•		•	•		
swprintf	23	•		•	•		
swscanf	23	•		•	•		
symlink	21		•		•		
sync	18		•		•		
sysinfo	26		•				
syslog	26	•			•		
system	4	•		•	•		
tan	24	•		•	•		
tanh	24	•		•	•		
tcdrain	33	•			•		
tcflow	33	•			•		
tcflush	33	•			•		
tcgetattr	33	•			•		
tcgetpgrp	33	•			•		
tcsendbreak	33	•			•		
tcsetattr	33	•			•		
tcsetpgrp	33	•			•		
tdelete	17	•			•		
tdestroy	17	•				•	
telldir	20	•			•		
tempnam	20	•			•		
textdomain	27	•				•	
tfind	17	•			•		
tgamma	27	•				•	
time	25		•	•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
timeradd	9	•				•	
timerclear	9	•				•	
timerisset	9	•				•	
timersub	9	•				•	
times	9		•		•		
tmpfile	20	•		•	•		
tmpnam	20	•		•	•		
tmpnam_r	20	•			•		
toascii	23	•			•		
tolower	23	•		•	•		
toupper	23	•		•	•		
towlower	23	•		•	•		
towupper	23	•		•	•		
trunc	24	•			•		
truncate	21		•		•		
tsearch	17	•			•		
ttyname	33	•			•		
ttyname_r	33	•			•		
twalk	17	•			•		
tzset	25	•			•		
umask	21		•		•		
uname	26		•		•		
ungetc	10	•		•	•		
ungetwc	23	•		•	•		
unlink	20		•		•		
unlockpt	33	•			•		
unsetenv	3	•					
updwtmp	26	•				•	
usleep	9	•			•		
ustat	26		•				•
utime	21		•		•		
utimes	21		•		•		
utmpname	26	•					
vfscanf	10	•					
vfprintf	10	•		•	•		
vfwprintf	23	•		•	•		

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
vfwscanf	23	•		•	•		
vprintf	10	•		•	•		
vscanf	10	•					
vsnprintf	10	•			•		
vsprintf	10	•		•	•		
vsscanf	10	•					
vswprintf	23	•		•	•		
vswscanf	23	•		•	•		
vwprintf	23	•		•	•		
vwscanf	23	•		•	•		
wait	5		•		•		
wait3	5		•		•		
wait4	5		•				
waitpid	5		•		•		
wcrtomb	23	•		•	•		
wcscasecmp	23	•				•	
wcscat	23	•		•	•		
wcschr	23	•		•	•		
wcscmp	23	•		•	•		
wcscoll	23	•		•	•		
wcscpy	23	•		•	•		
wcscspn	23	•		•	•		
wcslen	23	•		•	•		
wcsncasecmp	23	•				•	
wcsncat	23	•		•	•		
wcsncmp	23	•		•	•		
wcsncpy	23	•		•	•		
wcsnlen	23	•				•	
wcsnrtombs	23	•				•	
wcspbrk	23	•		•	•		
wcsrchr	23	•		•	•		
wcsrtombs	23	•		•	•		
wcsspn	23	•		•	•		
wcsstr	23	•		•	•		
wcstod	23	•		•	•		
wcstof	23	•		•		•	

Nom	Chapitre	Fonction	Appel-système	Iso C9X	SUSv3	Gnu	Obsolète
wcstok	23	•		•	•		
wcstol	23	•		•	•		
wcstold	23	•		•		•	
wcstoll	23	•		•		•	
wcstombs	23	•		•	•		
wcstoul	23	•		•	•		
wcstoull	23	•		•		•	
wcsxfrm	23	•		•	•		
wctob	23	•			•		
wctomb	23	•			•		
wmemchr	23	•		•	•		
wmemcmp	23	•		•	•		
wmemcpy	23	•		•	•		
wmemmove	23	•		•	•		
wmemset	23	•		•	•		
wordexp	20	•			•		
wordfree	20	•			•		
wprintf	23	•		•	•		
wscanf	23	•		•	•		
write	19		•		•		
writev	19		•		•		
y0	24	•			•		
y1	24	•			•		
yn	24	•			•		

Annexe 3

Bibliographie

Standards

On trouvera la norme **SUSv3** (*Single Unix Specifications version 3*) sur le site de l'association *Open Group* :

http://www.opengroup.org/

Les **RFC** (*request for comments*) sont disponibles sur de nombreux sites. Entre autres celui de l'IETF :

http://www.ietf.org/rfc.html

Livres et articles

[ANDRÉ 1996] Jacques André – *Iso Latin-1, norme de codage des caractères européens ? Trois caractères français en sont absents !* – Cahiers GUTenberg numéro 25 – novembre 1996. *http://www.gutenberg.eu.org/pub/GUTenberg/*.

[BACH 1989] Maurice J. Bach – *Conception du système Unix* – Masson, Prentice-Hall, 1989 (titre original *The Design of the UNIX Operating System*).

[BECKETT 1990] Brian Beckett – *Introduction aux méthodes de la cryptologie* – Masson, 1990 (titre original *Introduction to Cryptology*).

[BENTLEY 1989] Jon Bentley – *Programming Pearls* – Addison-Wesley, 1989.

[BLAESS 2000] Christophe Blaess – *Renforcer une station Linux face à un risque d'arrêt intempestif* – Linux Magazine France numéro 20, septembre 2000. (disponible sur *http://www.blaess.fr/christophe/*).

[BLAESS 2000a] Christophe Blaess – *Sécuriser l'accès physique à une station Linux* – Linux Magazine France numéro 21, octobre 2000. (disponible sur *http://www.blaess.fr/christophe/*).

[BLAESS 2001] Christophe Blaess – *Langages de scripts sous Linux* – Eyrolles, 2001.

[BLAESS 2001a] Christophe Blaess, Christophe Grenier, Fréderic Raynal – *Eliminer les failles de sécurité dès le développement d'une application* – Linux Magazine France numéros 23 à 28, décembre 2000 à mai 2001. (disponible sur *http://www.blaess.fr/christophe/*).

[BLAESS 2003] Christophe Blaess – *Linux, histoire d'un noyau* – Linux Magazine France , Hors-série numéro 16, septembre 2003. (disponible sur *http://www.blaess.fr/christophe/*).

[BOURNE 1991] Steve Bourne – *Le système Unix* – InterÉditions, 1991 (titre original *The Unix System*).

[BUTENHOF 1997] David Butenhof – *Programming with POSIX Threads* – Addison Wesley, 1997.

[CARD 1997] Rémy Card, Éric Dumas et Franck Mével – *Programmation Linux 2.0* – Eyrolles, 1997.

[CASSAGNE 1998] Bernard Cassagne – Introduction au langage C – *ftp://ftp.imag.fr/pub/archive/doc-unix/C/Introduction_ANSI_C.pdf*.

[CHESWICK 1991] Bill Cheswick – *An Evening With Berferd, in Which a Hacker Is Lured, Endured, and Studied* – *http://cm.bell-labs.com/who/ches/papers/.*

[FERRERO 1993] Alexis Ferrero – *Les réseaux Ethernet* – Addison-Wesley, 1993.

[FRISCH 2003] Ælen Frisch – *Les bases de l'administration système* – O'Reilly & Associates, troisième édition 2003 (titre original *Essential System Administration*).

[GALLMEISTER 1995] Bill O. Gallmeister – Posix.4 Programming for the Real World – O'Reilly & Associates, 1995.

[Garfinkel 1995] Simson Garfinkel – *PGP Pretty Good Privacy* – O'Reilly & Associates, 1995.

[GOLDSCHLAGER 1989] Les Goldschlager et Andrew Lister – *Informatique et algorithmique* – InterÉditions, 1989 (titre original *Computer Science : a Modern Introduction*).

[HERNERT 2003] Patrice Hernert – *Les algorithmes* – collection « Que sais-je ? », Presses universitaires de France, deuxième édition 2003.

[KERNIGHAN 1994] Brian W. Kernighan et Denis M. Ritchie – *Le langage C* – Masson, Prentice-Hall, 1994. Traduction française par Jean-François Groff et Éric Mottier (titre original *The C Programming Language*).

[KIRCH 2001] Olaf Kirch – *Administration réseau sous Linux* – O'Reilly & Associates, deuxième édition 2001 (titre original *Linux Network Administrator's Guide*).

[KNUTH 1973a] Donald E. Knuth – *The Art of Computer Programming – Fundamental Algorithms* – volume 1, Addison-Wesley Publishing Company, 1973.

[KNUTH 1973b] Donald E. Knuth – *The Art of Computer Programming – Seminumerical Algorithms* – volume 2, Addison-Wesley Publishing Company, 1973.

[KNUTH 1973c] Donald E. Knuth – *The Art of Computer Programming – Sorting and Searching* – volume 3, Addison-Wesley Publishing Company, 1973.

[KŒNIG 1992] Andrew Kœnig – *Pièges du langage C* – Addison-Wesley, 1992 (titre original *C Traps and Pitfalls*).

[LEVINE 1994] John R. Levine, Tony Mason et Doug Brown – *lex & yacc* – O'Reilly & Associates, 1994 (titre original *lex & yacc*).

[LEWINE 1994] Donald Lewine – *Posix Programmer's Guide* – O'Reilly & Associates, 1994.

[LOUKIDES 1997] Mike Loukides et Andy Oram – *Programmer avec les outils Gnu* – O'Reilly & Associates, 1997 (titre original *Programming With Gnu Software*).

[LOVE 2003] Robert Love – *Linux Kernel Development* – Developer's Library, 2003.

[MCCONNELL 1994] Steve McConnell – *Programmation professionnelle* – Microsoft Press, 1994 (titre original *Code Complete*).

[MAGUIRE 1995] Steve Maguire – *L'art du code* – Microsoft Press, 1995 (titre original *Writing Solid Code*).

[MINOUX 1986] Michel Minoux et Georges Bartnik – *Graphes, algorithmes, logiciels* – Dunod informatique, 1986.

[NELSON 1994] Mark Nelson – *Communications série, guide du développeur C++* – Dunod, 1994 (titre original *Serial Communication, A C++ Developper's Guide*).

[NEWHAM 1995] Cameron Newham et Bill Rosenblatt – *Le shell Bash* – O'Reilly & Associates, 1995 (titre original *Learning the Bash Shell*).

[NICHOLS 1996] Bradford Nichols, Dick Buttlar et Jacqueline Proulx Farrell – *Pthreads Programming* – O'Reilly & Associates, 1996.

[ODIN 2000] David Odin – *Programmation Linux avec GTK+* – Eyrolles, 2000.

[PRESS 1993] William H. Press *et al.* – *Numerical Recipes in C, the Art of Scientific Computing* – Cambridge University Press, 1993.

[RIFFLET 1995] Jean-Marie Rifflet – *La programmation sous Unix* – Édiscience International, 1995.

[RUBINI 2002] Alessandro Rubini – *Pilotes de périphériques sous Linux* – O'Reilly & Associates, deuxième édition 2002 (titre original *Linux Device Drivers).*

[STEVENS 1990] W. Richard Stevens – *Unix Network Programming* – Prentice-Hall, 1990.

[STEVENS 1993] W. Richard Stevens – *Advanced Programming in the Unix Environment* – Addison-Wesley, 1993.

[TANENBAUM 1997] Andrew S. Tanenbaum et Albert S. Woodhull – *Operating Systems, Design and Implementation* – Prentice-Hall, seconde édition, 1997.

[TORVALDS 2001] Linux Torvalds, David Diamond – *Il était une fois Linux* – Osman Eyrolles Multimédia, 2001 (titre original *Just For Fun*).

[WELSH 2003] Matt Welsh, Lar Kaufman, Terry Dawson, Matthias Dalheimer – *Le système Linux* – O'Reilly & Associates, quatrième édition, 2003 (titre original *Running Linux*).

Documents informatiques

Voici quelques sujets de documentation accessible avec la commande `info` lorsque le paquetage correspondant est installé. Nous avons sélectionné des thèmes particulièrement utiles pour le développeur :

- `Standards` : normes de codage pour les logiciels Gnu ;
- `Gcc` : documentation du compilateur C Gnu ;

- `Libc` : documentation de la bibliothèque GlibC ;
- `Cpp` : documentation du préprocesseur C Gnu ;
- `As` : l'assembleur Gnu ;
- `Ld` : documentation de l'éditeur de liens Gnu ;
- `Binutils` : ensemble d'outils permettant la manipulation des fichiers exécutables et objet ;
- `Gdb` : documentation du débogueur Gnu ;
- `Gprof` : documentation du profileur Gnu ;
- `Make` : documentation du constructeur d'application Gnu ;
- `Automake` : documentation de l'outil de création des fichiers *Makefile* ;
- `Autoconf` : système d'élaboration des scripts de configuration pour les sources d'un logiciel ;
- `Grep` : documentation de l'outil de recherche `grep`
- `Indent` : documentation de l'outil de mise en forme Gnu ;
- `Readline` : documentation de la bibliothèque Gnu *Readline*.

Les documents *HOW-TO* Linux couvrent un grand nombre de sujets et sont pour la plupart disponibles en version française. On les trouve en version originale sur le site du *Linux Documentation Project* : *http://www.tldp.org/* et en version française sur *http://www.traduc.org/*.

- Adv-Routing-HOWTO : routage avancé et gestion du trafic réseau ;
- Bridge+Firewall+DSL-mini-HOWTO : configuration d'une passerelle réseau avec firewall pour une liaison type ADSL ;
- C++-dlopen-mini-HOWTO : chargement dynamique de fonctions en C++ ;
- Config-HOWTO : trucs et astuces pour personnaliser une station Linux ;
- Ethernet-HOWTO : configuration des cartes réseaux ;
- Francophones-HOWTO : configuration d'un système Linux francophone ;
- From-PowerUp-To-Bash-Prompt-HOWTO : ce qu'il se passe depuis la mise sous tension jusqu'à la connexion utilisateur ;
- Glibc-Install-HOWTO : installation et configuration de `gcc` sur une machine Linux ;
- Glibc2-HOWTO : installation et configuration de la GlibC 2 ;
- Hardware-HOWTO : liste du matériel fonctionnant – ou ne fonctionnant pas – sous Linux ;
- Home-Network-mini-HOWTO : comment configurer une station Linux comme passerelle Internet pour un réseau personnel (ou de petite entreprise) ;
- IO-Port-Programming-mini-HOWTO : accès aux ports d'entrées-sorties physiques ;
- Kernel-HOWTO : configuration et compilation du noyau ;
- KernelAnalysis-HOWTO : fonctionnement interne du noyau ;
- NET3-4-HOWTO : configuration d'un réseau avec Linux ;
- Parallel-Processing-HOWTO : introduction à la programmation parallèle sous Linux ;
- RPM-HOWTO : utilisation de l'outil de paquetage rpm ;
- SCSI-Programming-HOWTO : programmation de drivers utilisant l'interface SCSI générique ;

- Security-HOWTO : considérations diverses à propos de la sécurité d'un système Linux ;
- Serial-HOWTO : configuration des lignes série ;
- Serial-Programming-HOWTO : accès logiciel aux ports série de la machine ;
- Software-Building-HOWTO : informations sur les meilleures méthodes de distribution de logiciels libres ;
- Software-Release-Practice-HOWTO : informations sur les dispositions légales pour la distribution de logiciels libres.

Index

Symboles

$$ 31
$? 94
/bin/ash 882
/bin/awk 76
/bin/chmod 33
/bin/date 201
/bin/df 696
/bin/find 578
/bin/gunzip 76
/bin/gzip 76
/bin/kill 131, 136, 137, 184, 266
/bin/ln 577, 579
/bin/login 59, 536, 682
/bin/mail 84, 86, 87
/bin/mkdir 543
/bin/mknod 584, 585
/bin/more 87, 228
/bin/mount 32, 67
/bin/nice 283
/bin/ping 32
/bin/ps 28-30, 103, 143, 265, 275
/bin/rmdir 543
/bin/sleep 94
/bin/stty 882
/bin/su 32, 682
/bin/sync 470, 807
/dev/cua0 897
/dev/log 582
/dev/printer 582
/dev/ptmx 891
/dev/pts 890
/dev/random 654, 656
/dev/ttyS0 897
/dev/urandom 654
/etc/fstab 415, 528, 690, 691, 694, 697
/etc/group 679-681
/etc/groups 37
/etc/hosts 833, 836
/etc/inetd.conf 516, 869
/etc/inittab 59, 468
/etc/mtab 692
/etc/network 837
/etc/passwd 37, 59, 94, 682
/etc/profile 59, 68
/etc/protocols 817
/etc/services 824, 869
/etc/shadow 683
/etc/shells 684
/etc/syslog.conf 704
/etc/termcap 874
/home/ftp 541
/proc 29, 537
/proc/mounts 692
/proc/self/cwd 537
/sbin/ifconfig 88, 814, 815
/sbin/ipfwadm 816
/sbin/route 815
/tmp 550, 552
/usr/bin/cdda2wav 48, 279
/usr/bin/cdrecord 279
/usr/bin/chsh 684
/usr/bin/gencat 716
/usr/bin/grep 420
/usr/bin/ipcrm 769
/usr/bin/ipcs 769
/usr/bin/kermit 33
/usr/bin/less 87, 133, 228
/usr/bin/localedef 713
/usr/bin/minicom 33, 572, 573
/usr/bin/mkfifo 749
/usr/bin/msgfmt 719
/usr/bin/mtrace 352, 353
/usr/bin/nohup 128
/usr/bin/perl 76
/usr/bin/renice 273
/usr/bin/top 275, 688
/usr/bin/uptime 688
/usr/bin/wish 76
/usr/locale/ 713
/usr/sbin/fuser 82, 517, 518, 538
/usr/sbin/inetd 854
/usr/share/locale/ 713, 719
/usr/src/linux/CREDITS 587
/var/adm/hostid 686
/var/log/messages 494, 706
__clone() 96
    définition 290
__free_hook 359, 361
__malloc_hook 359, 361
__realloc_hook 359, 361
_data 333
_end 333
_etext 333
_exit() 96, 97, 100, 126, 168
    définition 96
_IOFBF 487
_IOLBF 487
_IONBF 487
_NSIG 124
_PATH_UTMP 702
_PATH_WTMP 702
_POSIX_ASYNCHRONOUS_IO 798
_POSIX_MEMLOCK 371
_POSIX_MEMLOCK_RANGE 371
_POSIX_PRIORITY_SCHEDULING 275
_POSIX_REALTIME_SIGNALS 124, 133
_POSIX_SEMAPHORES 319

_POSIX_SYNCHRONIZED_IO 808
_POSIX_THREAD_ATTR_STACKADDR 299
_POSIX_THREAD_ATTR_STACKSIZE 299
_POSIX_THREAD_PRIORITY_SCHEDULING 299

A

ABDAY_1…ABDAY_7 733
ABMON_1…ABMON_12 733
abort() 81, 97, 99, 106, 125
    définition 97
abs() 648
    définition 648
accept() 302, 847
    définition 847
access() 572, 573
    définition 572
ACCOUNTING 698
acos()
    définition 636
acosf()
    définition 636
acosh()
    définition 638
acoshf()
    définition 638
acoshl()
    définition 638
acosl()
    définition 636
ACTION 459
Adams
    Douglas 427
addmntent()
    définition 694
adjtime()
    définition 664
adjtimex() 663
    définition 663
AF_INET 832, 840
AF_INET6 832, 840
AF_LOCAL 840
AF_UNIX 840
aio_cancel() 807
    définition 807
aio_error()
    définition 800
aio_fsync()
    définition 809
aio_read()
    définition 799
aio_return()
    définition 800
aio_suspend() 302, 805
    définition 805
aio_write()
    définition 799
alarm() 126, 139, 140, 172, 186, 195, 199
    définition 172
aléatoires
    générateurs ~ 654
alloca() 345, 346, 399
    définition 344
alphasort() 535
    définition 534
AM_STR 733
arbre binaire 450, 463
arc
    cosinus 636
    sinus 636
    tangente 636
argc 53, 54, 61, 62, 65
    définition 60
argument
    cosinus hyperbolique 638
    en ligne de commande 60, 90
    sinus hyperbolique 638
    tangente hyperbolique 638
argv 53, 54, 60-62, 64, 65, 75, 76, 79, 80
    définition 60
ARP 815
arrondi 646
as 7
Ascii 242, 403, 475, 625, 907
asctime() 667
    définition 667
asctime_r()
    définition 667
asin()
    définition 636
asinf()
    définition 636
asinh()
    définition 638
asinhf()
    définition 638
asinhl()
    définition 638
asinl()
    définition 636
assert() 97, 98, 99, 117, 122, 223, 343, 344
    définition 97
asynchrone
    entrée-sortie ~ 129, 504, 511, 796
atan()
    définition 636
atan2()
    définition 636
atan2f()
    définition 636
atan2l()
    définition 636
atanf()
    définition 636
atanh()
    définition 638
atanhf()
    définition 638
atanhl()
    définition 638
atanl()
    définition 636
atexit() 96, 100, 101, 103, 204
    définition 100
atof()
    définition 618
atoi()
    définition 618
atol()
    définition 618
atoll()
    définition 618
autoconf 19
automake 19

B

B0…B115200 900
basename() 411
    définition 411
bcmp()
    définition 394
bcopy()
    définition 393
Bessel
    fonctions de ~ 644

bibliothèque
DB Berkeley 604
DES 429
GDBM 589
GetText 718
libcap 48
mathématique 633
ncurses 23, 885
PCthreads 291
Readline 23
stdio 258, 264
big endian 820
bind() 843, 864
définition 842
bindtextdomain()
définition 719
bison 264, 414, 417
BOOT_TIME 698
Break 878
brk() 337
définition 334
BRKINT 878
broadcast 47, 863, 864
BS0 879
BS1 879
BSDLY 879
bsearch() 445, 448, 449
définition 445
btowc()
définition 631
BUFSIZ 487, 489
BUFSIZE 488
bzero()
définition 390
bzip2 20

C

cacos()
définition 643
cacosh()
définition 643
calloc() 336, 339, 340, 342, 343, 362, 373
définition 338
canonique 874
CAP_CHOWN 47, 573
CAP_DAC_OVERRIDE 47
CAP_DAC_READ_SEARCH 47
CAP_FOWNER 47
CAP_FSETID 47
CAP_IPC_LOCK 47, 369
CAP_IPC_OWNER 47, 760
CAP_KILL 47, 137
CAP_LEASE 47
CAP_LINUX_IMMUTABLE 47
CAP_NET_ADMIN 47
CAP_NET_BIND_SERVICE 47, 823
CAP_NET_BROADCAST 47
CAP_NET_RAW 47, 840
CAP_SETGID 39, 47, 682
CAP_SETPCAP 47
CAP_SETUID 47
CAP_SYS_ADMIN 47, 686, 697, 764
CAP_SYS_BOOT 47, 48
CAP_SYS_CHROOT 47, 541
CAP_SYS_MODULE 47
CAP_SYS_NICE 47, 48, 271, 275, 279, 284
CAP_SYS_PACCT 47
CAP_SYS_PTRACE 47
CAP_SYS_RAWIO 47, 48
CAP_SYS_RESOURCE 47, 133, 217, 764
CAP_SYS_TIME 47, 663
CAP_SYS_TTY_CONFIG 47
capacité 46
effective 46
possible 46
transmissible 46
capget()
définition 46
capset()
définition 46
casin()
définition 643
casinh()
définition 643
catan()
définition 643
catanh()
définition 643
catclose() 714
définition 716
catgets() 714
définition 715
catopen() 714
définition 714
cbc_crypt() 431, 432
définition 431
cbrt()
définition 641
cbrtf()
définition 641
cbrtl()
définition 641
cc1 7
cc1obj 7
cc1plus 7
ccos()
définition 643
ccosh()
définition 643
ceil()
définition 647
ceilf()
définition 647
ceill()
définition 647
cexp()
définition 643
cfgetispeed()
définition 900
cfgetospeed()
définition 900
cfmakeraw()
définition 883
cfsetispeed()
définition 900
cfsetospeed()
définition 900
chdir() 536, 538, 868
définition 536
Cheswick
Bill 541
chmod() 571
définition 571
chown() 573
définition 573
chroot() 47, 541
définition 541
clearenv()
définition 56
clearerr()
définition 491
CLK_TCK 213
CLOCAL 880, 897
clock() 212, 213
définition 212
clock_t 212, 213
CLOCKS_PER_SEC 212, 213

clog()
    définition 643
close() 302, 496, 513
    définition 496
closedir() 532
    définition 532
closelog()
    définition 704
close-on-exec 516-518
Code Warrior 21
comparison_fn_t 439
connect() 302
    définition 851
contrôle
    des jobs 41, 157, 158
copysign()
    définition 648
copysignf
    définition 648
copysignl
    définition 648
core 97, 98, 106, 125, 127, 129-131, 139, 168, 218, 223
cos()
    définition 635
cosf()
    définition 635
cosh()
    définition 637
coshf()
    définition 637
coshl()
    définition 637
cosl()
    définition 635
cpow()
    définition 643
cpp 7
CR0…CR1 879
CRDLY 879
CREAD 880
creat() 302, 496, 515, 582
    définition 493
CRTSCTS 880
crypt() 428-430
    définition 428
crypt_r()
    définition 430
cryptage 426
CS5…CS8 880
csin()
    définition 643
csinh()
    définition 643
CSIZE 880, 900
csqrt()
    définition 643
CSTOPB 880, 900
ctan()
    définition 643
ctanh()
    définition 643
ctermid()
    définition 886
ctime() 667, 676
    définition 667
ctime_r()
    définition 667
CURRENCY_SYMBOL 733
cuserid()
    définition 684
CVS 22

D

D_FMT 733
D_T_FMT 733
DATEMSK 673
datum 590, 594
DAY_1…DAY_7 733
DB 606
DB Berkeley 604
DB_BTREE 605
DB_HASH 605
DB_RECNO 605
DBM 587, 589, 590, 596-599, 604
dbm_clearerr()
    définition 599
dbm_close()
    définition 598
dbm_delete()
    définition 598
dbm_dirfno()
    définition 599
dbm_error()
    définition 599
dbm_fetch()
    définition 598
dbm_firstkey()
    définition 598
DBM_INSERT 599
dbm_nextkey()
    définition 598
dbm_open() 598
    définition 598
dbm_pagfno()
    définition 599
dbm_rdonly()
    définition 599
DBM_REPLACE 599
dbm_store()
    définition 598
dbmclose()
    définition 590
dbminit() 590
    définition 590
dbopen() 606
    définition 605
DBT 605
ddd 9
DEAD_PROCESS 698
DECIMAL_POINT 733
delete()
    définition 596
démon 43, 128, 867, 869
DES 428, 429, 430, 431
DES_DECRYPT 432
DES_ENCRYPT 432
DES_FAILED()
    définition 432
DES_HW 432
des_setparity()
    définition 433
DES_SW 432
descripteur 81, 227
    fermeture d'un ~ 96, 496
    ouverture d'un ~ 493
DESERR_BADPARAM 432
DESERR_HWERROR 432
DESERR_NOHWDEVICE 432
DESERR_NONE 432
diff 17
difftime()
    définition 675
Dijkstra
    Edsger 525, 772, 774
DIR 532
DISCARD 881
div()
    définition 649
drand48() 658
    définition 656

drand48_r()
définition 657
drem()
définition 649
dup() 514, 516, 518
définition 514
dup2() 515, 516, 517, 742, 887
définition 514

E

E2BIG 81, 115
EACCES 81, 115, 374, 521, 543, 761, 841
EADDRINUSE 844, 864
EAGAIN 30, 115, 121, 189, 320, 374, 503, 508, 521, 529, 785
EBADF 115, 517
EBUSY 115, 310, 314, 319, 549
ECANCELED 121
ecb_crypt() 431, 433
définition 431
ECHILD 106, 115
ECHO 880
ECHOCTL 880
ECHOE 880
ECHOK 880
ECHONL 880
Eclipse 21
ecvt() 623, 625
définition 622
ecvt_r()
définition 624
EDEADLK 115, 522
EDEADLOCK 310
EDOM 115, 650
EEXIST 115, 543, 761
EFAULT 82, 115, 117, 141
EFBIG 115, 133, 218, 508
egcs 7
EIDRM 761
EINPROGRESS 800
EINTR 115, 126, 139, 148, 190, 502, 503, 508, 521, 790
EINVAL 81, 115, 137, 141, 374, 517, 549, 840
EIO 81, 115, 131, 503, 508
EISDIR 81, 115, 549
ELIBBAD 81
ELOOP 81, 115
Emacs 4
EMFILE 81, 116
EMLINK 116
EMPTY 698
ENAMETOOLONG 81, 116
encrypt() 431
encrypt_r() 431
endfsent()
définition 691
endgrent()
définition 681
endhostent()
définition 836
endmntent()
définition 692
endnetent()
définition 837
endorder 453
endprotoent()
définition 819
endpwent()
définition 683
endservent()
définition 826
endusershell()
définition 684
endutent()
définition 699
endutxent()
définition 702
ENFILE 81, 116, 117
enm VISIT 453
ENODEV 116, 374
ENOENT 81, 116, 119, 761
ENOEXEC 81, 116
ENOLCK 116
ENOMEM 30, 81, 116, 117, 374, 761
ENOSPC 116, 508, 543, 761
ENOSYS 116, 757
ENOTBLK 116
ENOTDIR 81, 116
ENOTEMPTY 116, 543, 549
ENOTTY 116
ENTER 459
ENTRY 459, 461
environ 53, 54, 76, 77, 78
définition 52
environnement 51, 76, 80, 81, 559
ENXIO 116, 750
EOF 241, 242, 244, 245, 247, 249, 252, 253, 469, 477, 750, 881
EOL 881
EOL2 881
EOPNOTSUPP 428
EPERM 81, 116, 137, 310, 543
EPIPE 116, 130, 508, 746
EPROTONOSUPPORT 841
erand48()
définition 656
erand48_r)
définition 658
ERANGE 116, 117
ERASE 881
erf()
définition 643
erfc()
définition 643
erfcf()
définition 643
erfcl()
définition 643
erff()
définition 643
erfl()
définition 643
EROFS 116
erreur 114
fonction d'~ 643
errno 30, 81, 115, 130, 168, 231
définition 114
ESPIPE 116
ESRCH 116, 137
Ethernet 814
ETIMEDOUT 318
ETXTBSY 82, 116, 373, 374
EWOULDBLOCK 116, 529
EXDEV 116, 549
exec() 43, 46, 75, 84, 86, 158, 217, 279, 307, 308, 374, 516
execl() 75, 87
définition 80
execle() 75
définition 80
execlp() 75
définition 79
execv() 75
définition 78
execve() 75
définition 76
execvp() 75, 561
définition 78

exit() 44, 81, 95-97, 100, 101, 126, 129, 168, 266, 490
définition 95
EXIT_FAILURE 94
EXIT_SUCCESS 94, 96
exp()
définition 639
exp10()
définition 640
exp10f()
définition 640
exp10l()
définition 640
exp2()
définition 639
exp2f()
définition 639
exp2l()
définition 639
expf()
définition 639
expl()
définition 639
expm1()
définition 639
expm1f()
définition 639
expm1l()
définition 639
exponentielle 639

F

F_DUPFD 516
F_GETFD 517
F_GETFL 518, 519, 785
F_GETLK 520, 522
F_GETOWN 519, 797
F_GETSIG 519, 797
F_OK 572
F_RDLCK 520
F_SETFD 517
F_SETFL 518, 519, 784
F_SETLK 520, 521, 522
F_SETLKW 520, 521
F_SETOWN 519, 797
F_SETSIG 519, 797
F_UNLCK 520, 522
F_WRLCK 520
fabs()
définition 648
fabsf()
définition 648
fabsl()
définition 648
fchdir() 538
définition 536
fchmod() 571
définition 571
fchown() 573
définition 573
fclose() 86, 469, 470, 488, 496
définition 469
fcloseall() 469
définition 469
fcntl() 302, 374, 494, 516, 517, 520-522, 529, 784, 785, 796, 797
définition 516
fcvt() 623, 625
définition 622
fcvt_r()
définition 624
FD_CLOEXEC 517
FD_CLR()
définition 790
FD_ISSET()
définition 790
fd_set 790
FD_SET()
définition 790
FD_SETSIZE 790
FD_ZERO()
définition 790
fdatasync() 808
définition 808
fdopen() 472, 499, 552, 750
définition 471
feof()
définition 490
ferror() 491
définition 490
fetch() 595
définition 594
FF0 879
FF1 879
FFDLY 879
fflush() 470-472, 478, 485, 807, 876, 877
définition 469
fgetc() 244, 245, 247, 473, 480
définition 244
fgetgrent()
définition 681
fgetgrent_r()
définition 681
fgetpos() 477, 484
définition 484
fgetpwent()
définition 683
fgetpwent_r()
définition 683
fgets() 252, 259, 261, 473, 490
définition 251
fgetwc()
définition 628
fgetws()
définition 628
fichier
accès à un ~ 47
attributs d'un ~ 47, 567
temporaire 549
verrouillage d'un ~ 374
fifo
Voir tube nommé
FILE 227, 465, 467
fileno()
définition 491
filtre 228
FIND 459
find 17
finite()
définition 651
first_key()
définition 596
firstkey() 596
flex 264, 414, 417
flock() 520, 529
définition 529
floor() 647
définition 647
floorf()
définition 647
floorl()
définition 647
FLUSHO 880
flux
définition 227
fermeture d'un ~ 96, 469
ouverture d'un ~ 466, 471
positionnement dans un ~ 477
standard 227

fmod()
définition 649
fmodf()
définition 649
fmodl()
définition 649
FNM_CASEFOLD 553
FNM_FILE_NAME 553
FNM_LEADING_DIR 553
FNM_NOESCAPE 553
FNM_NOMATCH 553
FNM_PATHNAME 553
FNM_PERIOD 553
fnmatch() 553
définition 553
fopen() 227, 230, 467, 468, 471, 472, 491, 499, 551, 750
définition 467
fork() 29, 43, 44, 46, 51, 84, 86, 119-121, 158, 219, 279, 286, 291, 307, 308, 374, 379, 473, 518, 740, 868
définition 29
format
pour printf() 230
pour scanf() 255
fpos_t 484
fprintf() 86, 87, 168, 230, 238, 243, 473, 474, 475
définition 230
fputc() 241, 242, 244, 473, 480
définition 241
fputs() 473
définition 243
fputwc()
définition 628
fputws()
définition 628
FRAC_DIGITS 733
fread() 86, 139, 474, 475, 477, 490
définition 474
free() 167, 342, 343, 351, 352, 357, 359
définition 342
freopen() 472, 473, 514
définition 472
frexp()
définition 650
frexpf()
définition 650
frexpl()
définition 650
fscanf() 86, 264, 473, 474, 475
définition 255
fseek() 477, 478, 491, 511
définition 478
fseeko() 477, 478, 480
définition 480
fsetpos() 477, 478, 484, 491
définition 484
fstat() 569
définition 567
fstatfs() 696
définition 695
fsync() 302, 807, 808
définition 507
ftell() 477, 478, 479
définition 478
ftello() 477, 480
définition 480
ftime()
définition 663
ftok() 758
définition 758
ftruncate() 576
définition 575
ftw() 567
définition 563
FTW_CHDIR 564
FTW_D 564
FTW_DEPTH 564
FTW_DNR 564
FTW_DP 564
FTW_F 564
FTW_MOUNT 564
FTW_NS 564
FTW_PHYS 564
FTW_SL 564
FTW_SLN 564
fungetc() 473
fwprintf()
définition 629
fwrite() 86, 474, 475, 477
définition 473
fwscanf()
définition 629

G

gamma
fonction ~ 644
gcc 7, 97, 99
gcvt() 623, 625
définition 622
gdb 9, 131, 266, 268, 385
GDBM 587, 589, 590, 597-604
gdbm_close()
définition 602
gdbm_delete()
définition 602
gdbm_errno 601
gdbm_error 602
gdbm_exist()
définition 602
gdbm_fdesc()
définition 602
gdbm_fetch()
définition 602
GDBM_FILE 600, 601
gdbm_firstkey()
définition 602
GDBM_INSERT 602
GDBM_NEWDB 601
gdbm_nextkey()
définition 602
GDBM_NOLOCK 601
gdbm_open() 601
définition 601
GDBM_READER 601
gdbm_reorganize()
définition 602
GDBM_REPLACE 602
gdbm_setopt()
définition 602
gdbm_store() 602
définition 602
GDBM_SYNC 601
gdbm_sync()
définition 602
GDBM_WRCREAT 601
GDBM_WRITER 601
gdm_strerror() 603
get_current_working_dir_name() 537
définition 537
GETALL 776
getc() 245
définition 245
getchar()
définition 245
getcwd() 537, 538
définition 537

getdate() 670, 673, 675, 725
définition 672
getdate_err 672, 673
getdate_r() 672, 673
définition 672
getdomainname()
définition 686
getegid()
définition 37
getenv() 54, 56
définition 53
geteuid() 33
définition 32
getfsent()
définition 692
getfsfile()
définition 692
getfsspec()
définition 692
getgid()
définition 37
getgrent()
définition 681
getgrent_r()
définition 681
getgrgid() 680
définition 680
getgrgid_r() 680
définition 680
getgrnam() 680, 693
définition 680
getgrnam_r() 680
définition 680
getgroups() 38, 39
définition 38
gethostbyaddr()
définition 834
gethostbyaddr_r()
définition 834
gethostbyname()
définition 834
gethostbyname_r()
définition 834
gethostbyname2()
définition 834
gethostbyname2_r()
définition 834
gethostent()
définition 836
gethostid()
définition 686
gethostname() 685
définition 685
getitimer() 199
définition 200
getline() 253, 254, 261
définition 253
getlogin() 87, 90
définition 684
getlogin_r()
définition 684
getmntent() 696
définition 693
getmntent_r()
définition 693
GETNCNT 776
getnetbyaddr()
définition 837
getnetbyname()
définition 837
getnetent()
définition 837
getopt() 60, 62, 64, 65, 66
définition 62
getopt_long() 65, 66, 67, 69
définition 65
getopt_long_only() 65
définition 67
getpass()
définition 887
getpeername()
définition 845
getpgid() 41
définition 41
getpgrp() 138
définition 42
GETPID 776
getpid() 29, 138
définition 29
getppid()
définition 29
getpriority() 275
définition 273
getprotobyname()
définition 818
getprotobyname_r()
définition 820
getprotobynumber()
définition 818
getprotobynumber_r()
définition 820
getprotoent()
définition 819
getprotoent_r()
définition 820
getpt()
définition 890
getpwent()
définition 683
getpwent_r()
définition 683
getpwnam()
définition 683
getpwnam_r()
définition 683
getpwname() 559
getpwuid()
définition 683
getpwuid_r()
définition 683
getresgid()
définition 40
getrlimit() 133, 221, 345
définition 217
getrusage() 215
définition 215
gets() 249, 251
définition 249
getservbyname() 842
définition 825
getservbyname_r
définition 827
getservbyport() 842
définition 825
getservbyport_r
définition 827
getservent()
définition 826
getservent_r
définition 827
getsid()
définition 44
getsockname()
définition 844
getsockopt()
définition 863
getsubopt() 68, 69, 694
définition 67
gettext()
définition 718
gettimeofday() 318, 662, 663
définition 661

getuid() 33
    définition 32
getusershell()
    définition 684
getutent()
    définition 699
getutent_r()
    définition 699
getutid()
    définition 699
getutid_r()
    définition 699
getutline()
    définition 699
getutline_r()
    définition 699
getutxent()
    définition 702
getutxid()
    définition 702
getutxline()
    définition 702
GETVAL 776
getw() 477
    définition 477
getwc()
    définition 628
getwchar()
    définition 628
getwd() 537
    définition 537
GETZCNT 776
GID 37, 81
    effectif 37, 40
    réel 37, 81
    sauvé 40
gid_t 37
glob() 555, 556, 557, 558
    définition 555
GLOB_ABORTED 557
GLOB_ALTDIRFUNC 556
GLOB_APPEND 556
GLOB_BRACE 556
GLOB_DOOFS 556
GLOB_ERR 556, 557
GLOB_MARK 556
GLOB_NOCHECK 557
GLOB_NOESCAPE 557
GLOB_NOMAGIC 557
GLOB_NOMATCH 557
GLOB_NOSORT 557
GLOB_NOSPACE 557
GLOB_PERIOD 557
glob_t 555, 557
GLOB_TILDE 557
GLOB_TILDE_CHECK 557
globfree() 557, 558
    définition 555
gmtime()
    définition 665
gmtime_r()
    définition 665
Gnome 6, 24
Gnu
    application 64
    extension 231, 253
goto 126, 305, 306
GPG 431, 435
gprof 13
grantpt()
    définition 891
grep 17, 94
groupe
    d'utilisateurs 37, 41, 679
    de processus 37, 41, 44, 45, 887
    multicast 865
    supplémentaire 38, 39, 40, 47
GTK 25
gzip 20

H

h_errno 837
hasmntopt()
    définition 694
hcreate()
    définition 458
hcreate_r()
    définition 458
hdestroy()
    définition 459
hdestroy_r()
    définition 458
herror()
    définition 838
Hoare C. 446
HOME 58, 59, 60, 467, 536, 559
HOST_NOT_FOUND 837
hostent
    définition 834
hsearch() 459
    définition 459
hsearch_r() 459
    définition 458
hstrerror()
    définition 838
htonl()
    définition 822
htons()
    définition 822
HUGE_VAL 652
HUPCL 880
hypot()
    définition 637
hypotf()
    définition 637
hypotl()
    définition 637
HZ 197, 199

I

ICANON 880
ICMP 816, 863
ICRNL 878
IEEE 754 650, 653
IEEE 854 653
IEXTEN 880
IFS 60, 86
IGNBRK 878
IGNCR 878
IGNPAR 878
imake 19
IMAXBEL 878
INADDR_ANY 843, 850
INADDR_BROADCAST 829
INADDR_NONE 828, 829
indent 15
index()
    définition 411
inet_addr()
    définition 828
inet_aton()
    définition 828
inet_lnaof()
    définition 830
inet_netof() 831
    définition 830
inet_ntoa()
    définition 828
inet_ntop()
    définition 832
inet_pton()
    définition 832

inetd 516, 867
infinis 651
info 11
init 29, 30, 58, 96, 105, 130, 137, 266
INIT_PROCESS 698
initgroups()
    définition 682
initstate()
    définition 656
INLCR 878
INPCK 878
INT_CUR_SYMBOL 733
INT_FRAC_DIGITS 733
INT_MAX 232
internationalisation 625, 709
INTR 881
ioctl() 874
IP 814, 815
    ng 815
    v4 828
    v6 828, 832
IP_ADD_MEMBERSHIP 865, 867
IP_DROP_MEMBERSHIP 867
IP_MULTICAST_IF 867
IP_MULTICAST_LOOP 867
IP_MULTICAST_TTL 867
IPC Système V 757
IPC_CREAT 760, 770
IPC_EXCL 760, 770
IPC_NOWAIT 762, 763, 774
IPC_PRIVATE 758, 760
IPC_RMID 764, 771, 776
IPC_SET 764, 771, 776
IPC_STAT 764, 771, 776
IPPROTO_ICMP 840
IPPROTO_IP 863, 867
IPPROTO_RAW 840
IPPROTO_TCP 863, 867
isalnum()
    définition 615
isalpha()
    définition 615
isascii()
    définition 615
isatty()
    définition 885
isblank()
    définition 615
iscntrl()
    définition 615
isdigit()
    définition 615
isgraph()
    définition 615
ISIG 880
isinf()
    définition 651
islower()
    définition 615
isnan()
    définition 650
ISO 8859-1 625, 907
ISO 8859-15 625, 907
ISO_9660 543
ISO-4217 727
ISO-8859-1 403
isprint()
    définition 615
ispunct()
    définition 615
isspace() 260
    définition 615
ISTRIP 878
isupper()
    définition 615
iswalnum()
    définition 627
iswalpha()
    définition 627
iswblank()
    définition 627
iswcntrl()
    définition 627
iswdigit()
    définition 628
iswgraph()
    définition 628
iswlower()
    définition 628
iswprint()
    définition 628
iswpunct()
    définition 628
iswspace()
    définition 628
iswupper()
    définition 628
iswxdigit()
    définition 628
isxdigit()
    définition 615
ITIMER_PROF 199, 200, 204
ITIMER_REAL 199-201, 204
ITIMER_VIRTUAL 199, 200, 204
IUCLC 878
IXOFF 879
IXON 879

J

j0()
    définition 644
j0f()
    définition 644
j0l()
    définition 644
j1()
    définition 644
j1f()
    définition 644
j1l()
    définition 644
jiffies 213
jn()
    définition 644
jnf()
    définition 644
jnl()
    définition 644
jrand48()
    définition 657
jrand48_r()
    définition 658

K

Kde 6, 24
KDevelop 21
Kernighan
    Brian W. 15, 230, 395
key_t 758, 770
kflushd 808
KILL 881
kill() 137, 138, 139, 179, 181, 184
    définition 136
killpg() 138
    définition 138

L

L_ctermid 886
L_cuserid 684

L_INCR 478
L_SET 478
L_tmpnam 549
L_XTND 478
labs() 648
    définition 648
LANG 403, 711, 713
LC_ALL 403, 711, 713
LC_COLLATE 711
LC_CTYPE 711
LC_MESSAGES 711
LC_MONETARY 711
LC_NUMERIC 711
lchown() 573
    définition 573
lclint 14, 238
lcong48()
    définition 657
lcong48_r()
    définition 658
ld 7, 251
ldexp()
    définition 650
ldexpf()
    définition 650
ldexpl()
    définition 650
ldiv()
    définition 649
ldiv_t 649
leader 43
leaf 453, 454, 456
lesstif 24
lex
    Voir flex
lfind() 440-442, 444
    définition 440
lgamma()
    définition 644
lgammaf()
    définition 644
lgammal()
    définition 644
lien
    physique 577
    symbolique 579
limites
    d'un processus 217
link() 577, 582
    définition 577
LinuxThreads
    bibliothèque 291
lio_listio() 804
    définition 804
LIO_NOP 804
LIO_NOWAIT 804
LIO_READ 804
LIO_WAIT 804
LIO_WRITE 804
listen() 847
    définition 847
little endian 820
llabs() 648
    définition 648
lldiv()
    définition 649
lldiv_t 649
LNEXT 881
localeconv() 732
    définition 729
localisation 60, 119, 122, 135, 237, 403, 405, 412, 616, 621, 667, 670, 709
localtime() 676
    définition 665
localtime_r()
    définition 665
LOCK_EX 529
LOCK_NB 529
LOCK_SH 529
lockf() 302
log()
    définition 640
LOG_ALERT 705
LOG_AUTH 705
LOG_AUTHPRIV 705
LOG_CONS 704
LOG_CRIT 705
LOG_CRON 705
LOG_DAEMON 704
LOG_DEBUG 705
LOG_EMERG 705
LOG_ERR 705
LOG_FTP 705
LOG_INFO 705
LOG_KERN 704
LOG_LOCAL0 705
LOG_LPR 705
LOG_MAIL 704
LOG_NDELAY 704
LOG_NEWS 705
LOG_NOTICE 705
LOG_PERROR 704
LOG_PID 704
LOG_SYSLOG 705
LOG_USER 704
LOG_UUCP 705
LOG_WARNING 705
log10()
    définition 641
log1p()
    définition 640
log1pf()
    définition 640
log1pl()
    définition 640
log2p()
    définition 641
log2pf()
    définition 641
log2pl()
    définition 641
logarithme 640
logf()
    définition 640
LOGIN_PROCESS 698
logl()
    définition 640
LOGNAME 59, 60
LONG_MAX 213
longjmp() 127, 129, 170, 172
lrand48() 658
    définition 656
lrand48_r()
    définition 658
ls 85, 86
lsearch() 441, 442
    définition 440
lseek() 511, 513
    définition 511
lstat() 557, 569, 573
    définition 567

M

M_MAP_THRESHOLD 350
M_MMAP_MAX 351
M_MMAP_THRESHOLD 351
M_PI 634
M_TOP_PAD 351
M_TRIM_THRESHOLD 351
MAC 814

main() 53, 60, 62, 81, 93, 95-97, 100, 266, 488
make 18
Makefile 18
malloc() 60, 167, 332, 334, 336, 337, 339, 341-344, 346, 351, 352, 354, 355, 359, 373
    définition 332
MALLOC_CHECK_ 357, 358
MALLOC_MMAP_MAX_ 351
MALLOC_MMAP_THRESHOLD_ 351
MALLOC_TOP_PAD_ 351
MALLOC_TRACE 352, 354, 363
MALLOC_TRIM_THRESHOLD_ 351
mallopt()
    définition 351
mand 528, 694
MAP_ANON 373
MAP_ANONYMOUS 373
MAP_DENYWRITE 373, 374
MAP_FIXED 373, 374
MAP_GROWSDOWN 373
MAP_PRIVATE 372-374
MAP_SHARED 372-374, 770
masque
    umask 585
MAX_CANON 875
MAX_MAP_COUNT 374
MAXPATHLEN 540
MB_CUR_MAX 630, 631
MB_LEN_MAX 630
mblen()
    définition 631
mbrlen()
    définition 631
mbrtowc() 631
    définition 631
mbsnrtowcs()
    définition 632
mbsrtowcs()
    définition 631
mbstate_t 630
mbstowcs()
    définition 631
mbtowc()
    définition 630
mcheck() 354, 355
    définition 354
MCL_CURRENT 369
MCL_FUTURE 369, 370, 374
MD5 428, 429, 430
memccpy() 391
    définition 391
memchr() 396, 409, 410, 411
    définition 409
memcmp() 394
    définition 393
memcpy() 391, 392, 393
    définition 390
memfrob()
    définition 427
memmem() 410, 411
    définition 410
memmove() 393
    définition 392
mémoire
    insuffisante 117
    partagée 47, 770, 778
    protection de la ~ 382
    verrouillée 47, 367
    virtuelle 82, 333, 379
mempcpy()
    définition 391
memset() 340, 390
    définition 390
messages
    file de ~ 760
mkdir() 543, 582
    définition 543
mkfifo()
    définition 749
mknod() 584, 585, 749
    définition 582
mkstemp()
    définition 551
mktemp() 550, 551, 552
    définition 549
mktime() 676
    définition 666
mlock() 47, 218, 369-371
    définition 368
mlockall() 369-371, 374
    définition 368
mmap() 336, 337, 339, 351, 373-375, 379, 382, 770
    définition 371
mode_t 495
modf()
    définition 649
modff()
    définition 649
modfl()
    définition 649
MON_1…MON_12 733
MON_DECIMAL_POINT 733
MON_THOUSANDS_SEP 733
Motif 24, 168
mount()
    définition 697
mprobe() 355
    définition 355
mprotect() 382, 383
    définition 382
mrand48() 658
    définition 657
mrand48_r()
    définition 658
mremap()
    définition 381
MREMAP_MAYMOVE 382
MS_ASYNC 381
MS_INVALIDATE 381
MS_SYNC 381
MSG_DONTROUTE 858
MSG_EXCEPT 763
MSG_NOERROR 763
MSG_OOB 858
MSG_PEEK 858
msgctl() 759
    définition 764
msgget() 759-761
MSGMAX 760, 763
MSGMNB 764
MSGMNI 760
msgrcv() 139, 763
    définition 763
msgsnd() 139, 759, 763
    définition 762
msqid_ds 759
msync() 302
    définition 381
mtrace() 352, 354, 363, 366
    définition 352
multicast 47, 830, 865
multitâche 269
    préemptif 269
munlock() 368, 369, 371
    définition 368
munlockall() 368, 371
    définition 368

munmap() 375
définition 375
muntrace() 354
mutex 308
mutex_attr_t 311

N

N_CS_PRECEDES 733
N_SEP_BY_SPACE 733
N_SIGN_POSN 733
NAME_MAX 534
NaN 650, 651, 653
nanosleep() 197-199, 302
définition 197
NDBM 587, 589, 598-600
NDEBUG 9, 98, 99, 223, 344
Nedit 6
NEGATIVE_SIGN 733
NETDB_SUCCES 837
NEW_TIME 698
next_key()
définition 596
nextkey() 596
NFS 134
nftw() 563, 564
définition 563
NGROUPS 39
NIC 830
nice() 275
définition 271
NIS 600, 824
NL_CAT_LOCALE 715
nl_catd 715
nl_item 732
nl_langinfo() 732
définition 732
NL0 879
NL1 879
NLDLY 879
NLSPATH 715
nm 14
nmap 381
NNTP 823
NO_ADRESS 837
NO_RECOVERY 837
NOCLDSTOP 159
NODEFER 159
NOEXPR 733
NOFLSH 880
NPTL
bibliothèque 291
NR_OPEN 223
nrand48()
définition 656
nrand48_r()
définition 658
NSIG 124, 137
ntohl()
définition 822
ntohs()
définition 822
NTP 664

O

O_ACCMODE 518
O_APPEND 494, 495, 505, 506, 519, 785
O_ASYNC 797
O_CREAT 494-496, 498, 598
O_DIRECTORY 538
O_DSYNC 808
O_EXCL 494, 495, 498, 499, 551
O_NDELAY 495
O_NOCTTY 494
O_NONBLOCK 494, 495, 519, 750, 783-785
O_RDONLY 494, 518, 598
O_RDWR 494, 518, 598
O_RSYNC 808
O_SYNC 495, 785, 808
O_TRUNC 494, 495
O_WRONLY 494, 518
objdump 14
OCRNL 879
OFDEL 879
off_t 511
OFILL 879
OLCUC 879
OLD_TIME 698
on_exit() 96, 100, 102
définition 102
ONLCR 879
ONLRET 879
ONOCR 879
Open Group VI
open() 117, 303, 493-496, 499, 515, 518, 538, 573, 582, 750
définition 493
OPEN_MAX 469, 493
opendir() 532, 538, 557
définition 532
openlog()
définition 704
optarg 62, 64, 66, 68
définition 62
opterr 62
définition 62
optind 62
définition 62
optopt 62
définition 62
ordonnancement 27, 47, 200, 207, 265
des threads 299
FIFO 276
OTHER 277
priorité d'~ 46, 269
RR 277
temps-réel 48, 275
OSI 814

P

P_CS_PRECEDES 733
P_SEP_BY_SPACE 733
P_SIGN_POSN 733
P_tmpdir 550
PAGER 87
PARENB 880, 900
PARMRK 878
PARODD 880, 900
patch 18
PATH 59, 60, 76, 78-81, 85-87
PATH_MAX 537
pause() 139, 165, 166, 303
définition 165
pclose() 84, 86, 504
définition 86
PENDIN 880
perror() 119, 490
définition 119
personality()
définition 793
PGID 81, 137
PGP 431, 435
phtread_attr_t 297
phtread_mutexattr_t 309
PID 29, 81
pid_t 29
pipe() 493, 515, 842
définition 738
PIPE_BUF 747, 748, 750, 751
PM_STR 733

poll() 139, 303, 503, 788, 789, 794
définition 793
POLLERR 794
POLLHUP 794
POLLIN 793
POLLNVAL 794
POLLOUT 793
POLLPRI 794
popen() 84, 86, 87, 90, 130, 504
définition 86
port
réseau 47, 822
série 897
POSITIVE_SIGN 733
Posix VI
Posix.1b 123, 177, 191, 367
Posix.1e 46
POSIXLY_CORRECT 60
PostgreSQL 587
postorder 453, 456
pow()
définition 641
powf()
définition 641
powl()
définition 641
PPID 29, 31, 81
PPP 816
pread() 303, 504, 511, 803
preorder 453
printf() 230, 238, 239, 264, 710, 724, 725, 728
définition 238
PRIO_MAX 273
PRIO_MIN 273
PRIO_PGRP 273
PRIO_PROCESS 273
PRIO_USER 273
processus 27
états d'un ~ 265
fils 29, 31, 43, 44, 51, 96, 105, 108, 127, 158, 197, 266, 286, 379, 513, 738
leader 42-44, 96, 128
père 29, 31, 43, 44, 51, 93, 95, 96, 105, 127, 158, 197, 266, 286, 379, 513, 738
terminaison d'un ~ 93
PROT_EXEC 372, 382
PROT_NONE 372, 382, 383
PROT_READ 372, 382
PROT_WRITE 372, 374, 382, 383
protocole réseau 817
pselect()
définitions 793
psignal()
définition 134
pthead_equal() 296
PTHEAD_STACK_MIN 299
pthread_atfork() 307, 308
définition 307
pthread_attr_destroy()
définition 298
pthread_attr_getdetachstate()
définition 298
pthread_attr_getinheritsched()
définition 301
pthread_attr_getschedparam()
définition 300
pthread_attr_getschedpolicy()
définition 299
pthread_attr_getscope()
définition 300
pthread_attr_getstackaddr()
définition 299
pthread_attr_getstacksize()
définition 299
pthread_attr_init()
définition 297
pthread_attr_setdetachstate()
définition 298
pthread_attr_setinheritsched()
définition 301
pthread_attr_setschedparam()
définition 300
pthread_attr_setschedpolicy()
définition 299
pthread_attr_setscope() 300
définition 300
pthread_attr_setstackaddr()
définition 299
pthread_attr_setstacksize()
définition 299
pthread_attr_t 297
PTHREAD_CANCEL 292
pthread_cancel()
définition 301
PTHREAD_CANCEL_ASYNCHRONOUS 302, 303
PTHREAD_CANCEL_DEFERRED 302, 303
PTHREAD_CANCEL_DISABLE 301, 303
PTHREAD_CANCEL_ENABLE 301, 303
PTHREAD_CANCELED 301
pthread_cleanup_pop() 304, 305
définition 304
pthread_cleanup_push() 304, 305
définition 303
pthread_cond_broadcast()
définition 318
pthread_cond_destroy()
définition 314
pthread_cond_init()
définition 314
PTHREAD_COND_INITIALIZER 314
pthread_cond_signal() 315, 316, 318
définition 315
pthread_cond_t 314
pthread_cond_timedwait() 302
définition 318
pthread_cond_wait() 302, 315, 317, 318
définition 315
pthread_condattr_destroy()
définition 318
pthread_condattr_init()
définition 318
pthread_condattr_t 314
pthread_create() 291, 292, 297
définition 291
PTHREAD_CREATE_DETACHED 298
PTHREAD_CREATE_JOINABLE 298
pthread_detach() 296
définition 296
pthread_equal()
définition 291
pthread_exit() 295, 301, 312
définition 292
PTHREAD_EXPLICIT_SCHED 300
pthread_getschedparam()
définition 300
pthread_getspecific()
définition 324
PTHREAD_INHERIT_SCHED 300

pthread_join() 292, 295, 296, 302
définition 292
pthread_key_create()
définition 324
pthread_key_delete() 325
définition 324
pthread_key_t 324
pthread_kill() 326
définition 326
pthread_mutex_destroy()
définition 309
PTHREAD_MUTEX_ERRORCHECK 311
pthread_mutex_init() 309
définition 309
PTHREAD_MUTEX_INITIALIZER 309
pthread_mutex_lock() 310, 315
définition 310
PTHREAD_MUTEX_NORMAL 311
PTHREAD_MUTEX_RECURSIVE 311
pthread_mutex_t 309
pthread_mutex_trylock() 310
définition 310
pthread_mutex_unlock() 315
définition 310
pthread_mutexattr_destroy()
définition 311
pthread_mutexattr_gettype()
définition 311
pthread_mutexattr_init()
définition 311
pthread_mutexattr_settype()
définition 311
pthread_mutexattr_t 311
pthread_once() 306, 307, 325
définition 306
PTHREAD_ONCE_INIT 306
pthread_once_t 306
PTHREAD_SCOPE_PROCESS 300
PTHREAD_SCOPE_SYSTEM 300
pthread_self()
définition 296
pthread_setcancelstate()
définition 301
pthread_setcanceltype() 302
pthread_setschedparam() 299
définition 300
pthread_setspecific()
définition 324
pthread_sigmask()
définition 326
pthread_t 291, 297
pthread_testcancel() 302
définition 302
PThreads 289
ptrdiff_t 236
ptsname()
définition 891
ptsname_r()
définition 891
puissance 641
putc() 243
définition 242
putchar()
définition 242
putenv() 56
définition 56
putpwent()
définition 683
puts()
définition 243
pututline()
définition 700
pututxline()
définition 702
putw()
définition 477
putwc()
définition 628
putwchar()
définition 628
pwrite() 303, 511, 803
définition 505

Q

qecvt()
définition 624
qecvt_r()
définition 624
qfcvt()
définition 624
qfcvt_r()
définition 624
qgcvt() 625
définition 624
qsort() 407, 446, 448, 534
définition 446
Qt 25
quicksort 446
QUIT 881

R

R_OK 572
racine
carrée 641
cubique 641
raise()
définition 138
rand() 426
définition 655
RAND_MAX 655
rand_r()
définition 656
random()
définition 656
Rationnelles
expressions 419
rawmemchr()
définition 409
RCS 21
re_comp()
définition 426
re_exec()
définition 426
read() 131, 139, 140, 303, 502, 503, 504, 507, 508, 513, 787, 857
définition 502
readdir() 532-534, 536, 557
définition 532
readdir_r() 534
définition 533
readlink() 580
définition 580
readv() 303, 504
définition 503
realloc() 340-343, 352, 359
définition 340
realpath() 540, 541
définition 540
recherche 437
dichotomique 444, 463
séquentielle 440, 463
recv() 303, 845
définition 858
recvfrom() 303, 845
définition 857

recvmsg()
définition 858
REG_ESPACE 421
REG_EXTENDED 420, 426
REG_ICASE 420
REG_NEWLINE 420
REG_NOMATCH 421
REG_NOSUB 420, 421, 426
REG_NOTBOL 422
REG_NOTEOL 422
regcomp() 421, 422, 425, 535
définition 420
regerror() 421, 426
définition 421
regex_t 420, 421
regexec() 421, 422, 425, 535, 553
définition 421
regfree()
définition 422
regmatch_t 421
Régulières
expressions
Voir Rationnelles
remove() 549
définition 548
rename() 549, 694
définition 548
répertoire 47, 531
REPRINT 881
return() 81, 96, 97, 266
rewind() 478, 491
définition 478
rewinddir()
définition 534
rindex()
définition 411
rint()
définition 646
rintf()
définition 646
rintl()
définition 646
Ritchie
Denis M. 15, 230, 395
RLIM_INFINITY 217
rlim_t 222
RLIMIT_CORE 218
RLIMIT_CPU 218
RLIMIT_DATA 218, 334
RLIMIT_FSIZE 218, 509
RLIMIT_MEMLOCK 218, 374
RLIMIT_NOFILE 218, 223
RLIMIT_NPROC 119, 218
RLIMIT_RSS 218
RLIMIT_STACK 218, 345
rmdir() 543, 548, 549
définition 543
ROT-13 427
rpm 20
RSA 431
RSS 349
RUN_LVL 698
RUSAGE_CHILDREN 215
RUSAGE_SELF 215

S

S_IFBLK 583
S_IFCHR 583
S_IFIFO 582
S_IFREG 582
S_IRGRP 495, 496
S_IROTH 495, 496
S_IRUSR 495
S_IRWXG 495
S_IRWXO 495
S_IRWXU 495, 496
S_ISBLK() 569
S_ISCHR() 569
S_ISDIR() 569
S_ISFIFO() 569
S_ISGID 495, 543
S_ISLNK() 569, 579
S_ISREG() 569
S_ISSOCK() 569
S_ISUID 495
S_ISVTX 495, 543
S_IWGRP 495, 496
S_IWOTH 495
S_IWUSR 495
S_IXGRP 495, 496
S_IXOTH 495, 496
S_IXUSR 495
SA_INTERRUPT 152
SA_NOCLDSTOP 152
SA_NODEFER 153
SA_ONESHOT 153
SA_ONSTACK 153, 154
SA_RESETHAND 153
SA_RESTART 152, 172, 173, 193, 197
SA_SIGINFO 153, 181, 182
sbrk() 334, 336, 337, 340, 351
définition 334
scandir() 534, 535
définition 534
scanf() 255, 264
définition 255
SCCS 22
SCHED_FIFO 279, 284, 299
sched_get_priority_max()
définition 281
sched_get_priority_min()
définition 281
sched_getparam()
définition 282
sched_getscheduler()
définition 279
SCHED_OTHER 279, 281, 283, 299
SCHED_RR 279, 284, 299
sched_rr_get_interval()
définition 282
sched_setparam()
définition 282
sched_setscheduler() 287, 299
définition 284
sched_yield() 276, 277, 287
définition 286
seed48()
définition 657
seed48_r()
définition 658
SEEK_CUR 478, 511, 520
SEEK_END 478, 511, 520
SEEK_SET 478, 511, 520
seekdir()
définition 534
select() 132, 139, 196, 197, 303, 503, 509, 662, 788-790, 794, 796, 864
définition 789
sem_close
définition 320
sem_destroy()
définition 319
sem_getvalue()
définition 320
sem_init() 319
définition 319
sem_open
définition 320

sem_post() 320
définition 320
sem_t 319
sem_trywait()
définition 320
SEM_UNDO 774
SEM_VALUE_MAX 320
sem_wait() 303, 320
déinition 320
sémaphores
Posix.1b 319
Système V 772
semctl() 759
définition 773, 775
semget() 759
définition 773
semid_ds 759
struct 776
semop() 139, 759
définition 773
send() 303, 845
définition 858
sendmsg()
définition 858
sendto() 303, 845
définition 857
service réseau 822
session 43, 128
SETALL 776
setbuf() 469
définition 489
setbuffer()
définition 489
setdomainname()
définition 686
setegid()
définition 37
setenv() 56
définition 56
seteuid()
définition 34
setfsent()
définition 691
Set-GID 37, 40, 47, 81, 86, 249
setgid()
définition 37
Set-GID). 40
setgrent()
définition 681
setgroups() 682
définition 39
sethostent()
définition 836
sethostid()
définition 686
sethostname() 686
définition 685
setitimer() 126, 199, 201, 662
définition 199
setjmp() 170, 172
setkey() 431
setkey_r() 431
setlinebuf()
définition 489
setlocale() 404, 405, 413, 710, 712, 722
définition 721
setmntent()
définition 692
setnetent()
définition 837
setpgid() 43
définition 43
setpgrp()
définition 43
setpriority() 275
définition 273
setprotoent()
définition 819
setpwent()
définition 683
setregid()
définition 37
setresgid()
définition 40
setresuid()
définition 34, 36
setreuid() 34, 35
définition 34
setrlimit() 133, 334
définition 223
setservent()
définition 826
setsid() 44, 868
définition 44
setsockopt()
définition 863
setstate()
définition 656
settimeofday() 663
définition 663
Set-UID 33-35, 46, 47, 77, 81, 85, 86, 170, 249, 251
setuid() 34, 35
définition 34
setusershell()
définition 684
setutent()
définition 699
setutxent()
définition 702
SETVAL 776
setvbuf() 487, 489, 807, 853
définition 487
SHELL 59, 60
shell 29, 31, 43, 51, 55, 59, 61, 76, 77, 84-86, 93, 217, 228, 558, 684
bash 55, 60, 77, 93, 158, 169, 207, 214, 220, 221
ksh 55
tcsh 55, 77, 219, 221
SHM_LOCK 771
SHM_RDONLY
définition 771
SHM_RND
définition 771
SHM_UNLOCK 771
shmat() 759, 771
définition 770
shmctl() 759, 771
définition 770
shmdt() 759
définition 770
shmget() 759
définition 770
shmid_ds 759
SHMMAX 770
SHMMIN 770
shutdown() 857
définition 857
SI_ASYNCIO 182
SI_KERNEL 182
SI_MESGQ 182
SI_QUEUE 182, 186
SI_SIGIO 182
SI_TIMER 182, 186
SI_USER 182, 186
SID 43, 81
sig_atomic_t 167
SIG_BLOCK 162
SIG_DFL 141, 143, 152
SIG_ERR 141

SIG_IGN 141, 152
SIG_SETMASK 162
SIG_UNBLOCK 162
SIGABRT 97, 106, 108, 125, 126
sigaction() 140, 141, 151, 197, 325
    définition 151
sigaddset()
    définition 155
SIGALRM 126, 139, 140, 170, 172, 186, 199, 204
sigaltstack() 154
    définition 154
sigandset()
    définition 155
sigblock() 162
SIGBUS 127
SIGCHLD 86, 96, 108, 127, 139, 266, 848
SIGCLD 127
SIGCONT 96, 128, 131, 137, 143, 145, 152, 266
sigdelset()
    définition 155
sigemptyset()
    définition 155
SIGEV_NONE 800
SIGEV_SIGNAL 800
SIGEV_THREAD 800
sigfillset() 162
    définition 155
SIGFPE 127, 170
siggetmask() 162
SIGHUP 96, 128, 179
SIGILL 127-129, 170, 179
SIGINFO 134
SIGINT 86, 129, 156, 157, 158, 163, 245
siginterrupt() 148, 149
    définition 147
SIGIO 129, 130, 796
SIGIOT 125
sigisemptyset()
    définition 155
sigismember()
    définition 155
SIGKILL 130, 133, 138, 141-143, 151, 152, 162, 190, 218
siglongjmp() 126, 170, 346
    définition 170
SIGLOST 134
sigmask() 162
signal() 140, 159
    définition 141
signaux 44, 47, 81, 93, 97, 123, 151
    bloqués 138, 145, 158, 161, 179, 186, 189
sigorset()
    définition 155
sigpause()
    définition 167
sigpending()
    définition 163
SIGPIPE 130, 508, 745, 746, 750
SIGPOLL 129
sigprocmask() 161, 162, 165, 166, 190
    définition 161
SIGPROF 126
SIGPWR 134
sigqueue() 179, 181, 184
    définition 181
SIGQUIT 86, 130, 156, 157, 158, 160, 161
SIGRTMAX 124, 133, 178, 179
SIGRTMIN 124, 133, 178, 179
SIGSEGV 127, 168, 170, 179, 238, 249, 383, 385, 386, 396, 488
sigset_t 152, 155
    définition 155
sigsetjmp() 170
    définition 170
sigsetmask() 162
SIGSTKFLT 128
SIGSTOP 130, 131, 138, 141, 142, 143, 151, 266
sigsuspend() 167, 173, 189, 201, 303
    définition 166
SIGTERM 130, 131, 137, 160, 168
sigtimedwait() 189, 190, 303
    définition 189
SIGTRAP 131
SIGTSTP 131
SIGTTIN 131, 503
SIGTTOU 132
SIGUSR1 108, 132, 133, 162, 165
SIGUSR2 132, 133, 184
sigval_t 800
sigvec() 141
SIGVTALRM 126
sigwait() 303, 329
    définition 328
sigwaitinfo() 189, 190, 191, 303
    définition 189
SIGWINCH 133
SIGXCPU 133, 218
SIGXFSZ 133, 218, 508, 509, 511
sin()
    définition 635
sincos()
    définition 637
sincosf()
    définition 637
sincosl()
    définition 637
sinf()
    définition 635
sinh()
    définition 637
sinhf()
    définition 637
sinhl()
    définition 637
sinl()
    définition 635
size_t 236, 332
sizeof()
    définition 613
sleep() 126, 193, 195, 197, 199, 303
    définition 193
snprintf() 239
    définition 238
SO_BROADCAST 863, 864
SO_BSDCOMPAT 863, 864
SO_DEBUG 863
SO_DONTROUTE 863
SO_ERROR 863
SO_KEEPALIVE 863
SO_LINGER 863
SO_OOBINLINE 864
SO_RCVBUF 864
SO_RCVLOWAT 864
SO_RCVTIMEO 864
SO_REUSEADDR 864
SO_SNDBUF 864
SO_SNDLOWAT 864
SO_SNDTIMEO 864
SO_TYPE 864
SOCK_DGRAM 840, 845
SOCK_RAW 840
SOCK_STREAM 840

socket 130, 132, 839
raw 47
socket() 493, 515, 582
définition 839
socketpair()
définition 842
socklen_t 843
SOL_SOCKET 863
sous-option 67
speed_t 900
sprintf() 231, 239, 618, 622
définition 238
sqrt()
définition 641
sqrtf()
définition 641
sqrtl()
définition 641
srand()
définition 655
srand48()
définition 657
srand48_r()
définition 658
srandom()
définition 656
SS_DISABLE 154
SS_ONSTACK 154
sscanf() 261, 264, 618
définition 255
SSIZE_MAX 502
ssize_t 236, 502, 504
standard
bibliothèque ~ 230
entrée ~ 86, 228, 229
erreur ~ 90, 97, 228
sortie ~ 86, 87, 228, 229, 243
START 881
stat() 375, 557, 569, 573, 748
définition 567
statfs() 696
définition 695
statistiques
sur un processus 215
stderr 62, 228, 230, 238, 473, 485, 486, 514, 516
définition 228
STDERR_FILENO 493
stdin 245, 249, 264, 473, 514, 516
définition 228
STDIN_FILENO 493, 569
stdout 228, 230, 238, 242, 243, 473, 485, 514, 516
définition 228
STDOUT_FILENO 493, 569
stime() 663
définition 663
STOP 881
store() 590
définition 590
stpcpy() 398, 401
définition 398
stpncpy() 398, 401
définition 398
strace 13
strcasecmp() 403, 404, 438, 725
définition 403
strcasestr() 413
définition 412
strcat()
définition 400
strchr() 410, 411
définition 410
strcmp() 401, 402, 406, 407, 437
définition 401
strcoll() 405, 406, 725
définition 405
strcpy() 129, 395
définition 395, 397
strcspn()
définition 413
strdup() 399
définition 399
strdupa()
définition 399
strerror() 117-119, 122, 231, 705
définition 117
strerror_r()
définition 117
strfmon() 724, 726, 728
définition 725
strfry()
définition 426
strftime() 669, 670, 672, 676, 725, 728
définition 667
strip 14
strlen() 253, 395, 396, 401, 534
définition 395
strncasecmp() 403
définition 403
strncat() 400, 401
définition 400
strncmp() 56, 402
définition 402
strncpy() 392
définition 397
strndup() 399
définition 399
strndupa()
définition 399
strnlen() 396, 409
définition 396
strpbrk() 415
définition 413
strptime() 670, 671, 673, 675
définition 670
strrchr() 411
définition 410
strsep()
définition 416
strsignal() 135
définition 134
strspn() 413, 414
définition 413
strstr() 411
définition 411
strtod() 264
définition 620
strtof()
définition 620
strtok() 414, 416
définition 414
strtok_r()
définition 416
strtol() 264, 619
définition 619
strtold()
définition 620
strtoll()
définition 619
strtoul() 264
définition 619
strtoull()
définition 619
struct aiocb 798, 804
struct dirent 505, 532
struct file 505, 511, 518
struct file_operations 583
struct files_struct 505
struct flock 520, 522
struct fstab 691

struct group 680
struct hsearch_data 458
struct in_addr 828
struct in6_addr 828
struct inode 505
struct iovec 503
struct ip_mreq 865
struct ipc_perm 759
struct itimerval 200
struct lconv 729, 733
struct linger 863
struct option
    définition 65
struct passwd 682
struct pollfd 793
struct protoent 818, 821
struct rlimit 217, 221, 223
struct rusage 112, 215
    définition 111
struct sched_param 281, 284
struct sembuf 773
struct servent 825
struct shmid_ds 771
struct sigaction 181, 182
    définition 152
struct sigaltstack
    définition 154
struct sigevent 799
struct siginfo 181, 182
    définition 153
struct sockaddr 841
struct sockaddr_in 841
struct stat 488, 567-569, 573, 574, 576, 578, 579
struct statfs 695
struct sysinfo 688
struct termios 875, 897, 900
struct timeb 663
struct timebuf 576
struct timespec 190, 197, 282, 318
struct timeval 200, 662, 663, 664, 675, 789
    définition 112
struct timezone 662, 663
struct tm 664, 666, 668
struct tms 214
struct utimbuf 576
struct utmp 698
struct utmpx 701
struct utsname 687
strxfrm() 406, 725
    définition 406
stty 129, 245
stuct siginfo 185
SUSP 881
SUSv3 VI, 2
swprintf()
    définition 629
swscanf()
    définition 629
symlink() 582
    définition 579
sync() 470, 507
    définition 470
sys_errlist[]
    définition 121
sys_nerr 121
sys_siglist[]
    définition 134
sysinfo() 688
    définition 688
syslog() 705, 868
    définition 704
syslogd 707
system() 84-87, 303, 887
    définition 84

T

T_FMT 733
T_FMT_AMPM 733
TAB0…TAB1 879
TABDLY 879
table de hachage 456, 463
tan()
    définition 635
tanf()
    définition 635
tanh()
    définition 637
tanl()
    définition 635
tar 20
tcdrain() 303
    définition 877
tcflow()
    définition 877
tcflush()
    définition 877
tcgetattr() 876
    définition 875
tcgetpgrp() 45
    définition 887
TCIFLUSH 877
TCIOFF 877
TCIOFLUSH 877
TCION 877
Tcl/Tk 89
TCOFLUSH 877
TCOOFF 877
TCOON 877
TCP 814, 846
    présentation 816
TCP/IP 130, 132
TCP_MAXSEG 867
TCP_NODELAY 867
TCSADRAIN 875
TCSAFLUSH 875
TCSANOW 875
tcsendbreak()
    définition 878
tcsetattr() 875, 876
    définition 875
tcsetpgrp() 45
    définition 887
tdelete()
    définition 453
tdestroy()
    définition 453
telldir() 534
    définition 534
telnet 894
telnetd 894
tempnam() 60, 167, 549, 550, 551
    définition 549
temps
    d'exécution 81, 200, 207, 212
    d’exécution 126
temps-réel 47
    ordonnancement 275
    signaux 133, 138, 177
TERM 58, 59, 60
terminaison
    d’un processus 93, 131
    routine de ~ 97, 100
terminal 245, 873
    configuration du ~ 245
    de contrôle 43-45, 96, 128, 131, 887
    pseudo ~ 43, 890
textdomain()
    définition 719
tfind() 454
    définition 453

tgamma()
    définition 644
tgammaf()
    définition 644
tgammal()
    définition 644
THOUSANDS_SEP 733
Threads 289
threads 96
tilde 559
time() 199, 318, 661, 662, 663
    définition 195, 661
time_t 195, 576, 660, 661, 664, 675
TIME_WAIT 857
timeradd() 675
    définition 200
timerclear() 675
    définition 200
timerisset() 675
    définition 201
timersub() 675
    définition 201
times() 213, 214
    définition 213
TMPDIR 60, 550
tmpfile() 60, 96, 97, 100, 552
    définition 552
Tmpfs 319
tmpnam() 60, 549, 550, 551
    définition 549
tmpnam_r()
    définition 550
toascii() 617
    définition 617
tolower() 617
    définition 617
Torvalds
    Linus 18, 588
TOSTOP 132, 880
toupper() 617
    définition 617
towlower()
    définition 628
towupper()
    définition 628
tri 437
trunc()
    définition 647
truncate()
    définition 575
TRY_AGAIN 837
tsearch() 453
    définition 452
ttyname()
    définition 886
ttyname_r()
    définition 886
tube 130, 738
    nommé 749
twalk() 453, 454
    définition 453
TZ 60, 676
tzset() 60
    définition 676

U

UCHAR_MAX 241, 242, 244
UDP 814, 846
    présentation 816
UDP/IP 229
UID 32, 81
    effectif 32-35, 40, 46
    réel 32-34, 81
    sauvé 32-36
uid_t 32
UINT_MAX 232
umask 544, 545, 586
umask()
    définition 585
umount()
    définition 697
uname() 686
    définition 686
ungetc() 247, 258
    définition 247
ungetwc()
    définition 629
Unicode 625
union sigval 180, 800
Unix_98 VI
unlink() 548, 552, 577
    définition 546
unlockpt()
    définition 891
unsetenv() 56
    définition 56
updwtmp()
    définition 703
USER 59, 60
USER_PROCESS 698
usleep() 196, 197, 303
    définition 196
ustat()
    définition 697
UTF-8 630
utime() 576
    définition 576
utimes() 576
    définition 576
utmp 698
utmpname()
    définition 702

V

valeur absolue 648
verrouillage
    de fichier 520
vfprintf() 473
    définition 239
vfscanf() 473
    définition 255
vfwprintf()
    définition 629
vfwscanf()
    définition 629
Vi 4
VMIN 882, 884
vprintf() 239
    définition 239
vscanf()
    définition 255
vsnprintf() 239
    définition 239
vsprintf()
    définition 239
vsscanf()
    définition 255
vswprintf()
    définition 629
vswscanf()
    définition 629
VSZ 349
VT0 879
VT1 879
VTDLY 879
VTIME 882, 884
vwprintf()
    définition 629
vwscanf()
    définition 629

W

W_OK 572
waipid() 106
wait() 31, 99, 106, 108, 109, 111, 127, 139, 303
    définition 106
wait3() 106, 111, 112, 114, 215, 662
    définition 112
wait4() 106, 111, 114, 215
    définition 114
waitid() 114, 303
waitpid() 43, 99, 108, 109, 111, 114, 303
    définition 109
WCHAR_MAX 625
WCHAR_MIN 625
wchar_t 625, 632
WCOREDUMP()
    définition 106
wcrtomb() 631
    définition 631
wcscasecmp()
    définition 626
wcscat()
    définition 626
wcschr()
    définition 626
wcscmp()
    définition 626
wcscoll()
    définition 626
wcscpy()
    définition 625
wcscspn()
    définition 626
wcslen()
    définition 625
wcsncasecmp()
    définition 626
wcsncat()
    définition 626
wcsncmp()
    définition 626
wcsncpy()
    définition 626
wcsnlen()
    définition 625
wcsnrtombs()
    définition 632
wcspbrk()
    définition 626
wcsrchr()
    définition 626
wcsrtombs()
    définition 631
wcsspn()
    définition 626
wcsstr()
    définition 626
wcstod()
    définition 627
wcstof()
    définition 627
wcstok()
    définition 626
wcstol()
    définition 627
wcstold()
    définition 627
wcstoll()
    définition 627
wcstombs()
    définition 631
wcstoul()
    définition 627
wcstoull()
    définition 627
wcsxfrm()
    définition 626
wctob()
    définition 631
wctomb()
    définition 630
WEOF 625
WERASE 881
WEXITSTATUS()
    définition 106
WIFEXITED()
    definition 106
WIFSIGNALED()
    définition 106
WIFSTOPPED()
    définition 106
wint_t 625, 632
wmemchr()
    définition 626
wmemcmp()
    définition 627
wmemcpy()
    définition 627
wmemmove()
    définition 627
wmemset()
    définition 627
WNOHANG 109, 112
wordexp() 560, 561, 563
    définition 558
wordexp_t 558, 560
wordfree() 561, 563
    définition 558
wprintf()
    définition 629
WRDE_APPEND 560
WRDE_BADCHAR 561
WRDE_BADVAL 561
WRDE_CMDSUB 561
WRDE_DOOFFS 560
WRDE_NOCMD 560, 563
WRDE_NOSPACE 561
WRDE_REUSE 560
WRDE_SHOWERR 560
WRDE_SYNTAX 561
WRDE_UNDEF 560
write() 117, 130, 133, 303, 495, 505-509, 511, 513, 745, 746, 787, 857
    définition 505
writev() 303, 511
    définition 505
wscanf()
    définition 629
WSTOPSIG()
    définition 106
WTERMSIG()
    définition 106
wtmp 702
WUNTRACED 109, 111, 112

X

X_OK 572
X11 168
xargs 17
Xlib 24, 168, 354
Xm 24
xmkmf 19
Xt 24, 168, 354
XTABS 879
xxgdb 9

Y

y0()
    définition 645
y0f()
    définition 645
y0l()
    définition 645
y1()
    définition 645
y1f()
    définition 645
y1l()
    définition 645
yacc
    Voir bison
YESEXPR 733
yn()
    définition 645
ynf()
    définition 645
ynl()
    définition 645

Z

zombie 96, 103-106, 111, 127, 130, 266